PROGRAMOWANIE MIKROKONTROLERÓW AVR W JĘZYKU C

Opracowanie zawierające treści z różnych publikacji książkowych i internetowych odnośnie programowania mikrokontrolerów AVR w języku C

2007

SPIS TREŚCI

WPROWADZENIE

Niniejsze opracowanie ma służyć jako kompendium wiedzy o programowaniu mikrokontrolerów AVR w języku C. Przedstawione i zasygnalizowane problemy mają pomóc w zrozumieniu mechanizmu programowania procesorów AVR.
Przykłady i listingi zawarte w opracowaniu pozwalają na szybsze i lepsze poznanie elementów języka C. Należy nadmienić, że wykorzystywanym środowiskiem będzie środowisko ***WinXP + WinAVR + AVRdude***, czyli jak zdaje się autorowi opracowania - obecnie najpopularniejsze.

Autor usilnie pragnął zebrać jak największą ilość materiałów i w jego przekonaniu większość podstawowych informacji znajduje się na kolejnych stronach *„Programowanie mikrokontrolerów AVR w języku C" (PMAVR)*.

W niniejszym podręczniku zapewne znajdują się błędy. Główną ich przyczyną może być nieaktualność procedur i funkcji nie obsługiwanych przez najnowsze pakiety *WinAVR*, czy *AVRdude*.
Różni autorzy odpowiednich fragmentów opracowania sprawiają iż różne problemy przedstawione są w różny sposób. Czasem podejście nie jest optymalne. To użytkownik tego opracowanie musi sam dojść do swojego sposobu pisania programów, kierując się doświadczeniem swoim, ale również doświadczeniami różnych autorów tego podręcznika. Dzięki prezentacji różnych sposobów myślenia dochodzi się do większej wprawy w projektowaniu i uruchamianiu systemów mikroprocesorowych.

Podręcznik nie jest w żaden sposób wersją komercyjną. Wszelkie użyte materiały pochodzą ze stron internetowych i są powszechnie dostępne. Spis autorów tekstów, listingów znajduje się na końcu opracowania.
Celem powstania tego podręcznika było zebranie wiedzy potrzebnej na pierwsze samodzielne próby programowania mikrokontrolerów w języku C. Aktualna lista publikacji na ten temat nie jest duża. Jeśli z kolei chodzi o podręczniki w wersji polsko językcznej, możemy je policzyć na palach jędnej ręki. Stopień ich trudności może niekiedy przerazić początkującego. Dlatego powstała ta alternatywa, będąca małym kompedium wiedzy, nie tylko na temat programowania w języku C, ale także obsługi mikrokontrolerów AVR. Publikacja ta dostępna wyłącznie w formie pliku pdf, może stać się projektem *„open-sourcowym"* jeśli tylko będzie odpowiednie nią zainteresowanie.

Życze wielu udanych projektów mikroprocesorowych przede wszystkim w języku C.

autor opracowania
CK

AVRdude a programowanie w C

Zgodnie z oficjalną stroną projektu ***AVRdude*** rozszyfrowuje się jako ***AVR D****ownloader / **U**ploa**DE**r*. Jest to projekt *open source* czyli tzw. otwartego oprogramowania, umożliwiające każdemu dostęp do źródła programu, dzięki temu w tworzeniu bierze udział olbrzymia liczba osób.

Wracając do tematu, *AVRdude* jest to narzędzie do wgrywania, zgrywania, ładowania, manipulowania pamięciami ROM i EEPROM w mikrokontrolerach AVR, programowalnych w systemie ISP (*in-system programming*).

AVRdude jest programem uruchamianym w trybie tekstowym. Istnieją do niego nakładki graficzne umożliwiające łatwiejszą obsługę. My zajmiemy się jedynie aspektem pracy w trybie tekstowym. Opisane zostaną podstawowe funkcje. Cały opis dołączony jest do pakietu *WinAvr* (katalog doc).

„AVRdude a programowanie w C” - dlaczego taki tytuł tego rozdziału? Już tłumacze. AVRdude jest jednym z najpopularniejszych programów dla mikrokontrolerow AVR. Istnieje jeszcze np. *PonyProg*, łatwiejszy od *AVRdude*, ale niekiedy mniej intuicyjny. *AVRdude* pozwala na ładowanie do pamięci flash uC plików z rozszerzeniem hex (format *intel hex*) lub bin. Obsługuje bardzo wiele programatorów i to jest jego największa zaleta. Pracujący w trybie tekstowym *AVRdude* można zintegrować z różnego rodzaju oprogramowaniem, np. z *WinAvr* czy chociażby z *Bascomem*. To też przemawia za wyższością *AVRdude*.

Podstawowe wiadomości

AVRdude dołączony jest do oprogramowania *WinAvr,* znajduje się w katalogu bin. *WinAvr* można ściągnąć za darmo z internetu. Można również ściągnąć kody źródłowe *AVRdude* (źródła w języku C), a następnie je skompilować i uruchomić.

Aby uruchomić AVRdude należy wejść w tryb tekstowy (*WinXP, Uruchom→ cmd*). Następnie wpisać ścieżkę dostępu do AVRdude. Podstawowa składnia została zaprezentowana poniżej:

avrdude –p partno *options...*

Oczywiście pierwszy człon to wywołanie programu, zaś ***–p*** jest rozkazem dla *AVRdude* jaki procesor jest podłączony do programatora. Człon ***partno*** jest parametrem który znajduje się w plikach konfiguracyjnych programu i jest to nazwa procesora, np. dla *Atmegi8*.

avrdude –p m8 ...

W *„Dodatku”* umieszczono spis procesorów i identyfikatorów im przypisanym. Człon ***options...*** zawiera wiele opcji i możliwości których możemy użyć. Poniże znajduje się steszczenie używanych opcji:

- p *partno*
wytłumaczenie powyżej

- b *baudrate*
oznajmia szybkość transmisji połączenia szeregowego RS232

- c *programmer-id*
określa rodzaj używanego programatora. W *„Dodatku”* znajduje się lista dostępnych identyfikatorów.

-C ***config-file***
określa ścieżkę dostępu do pliku konfiguracyjnego dla avrdude
- D
wyłącza automatyczne kasowanie pamięci flash przy ładowaniu programu do procesora
-e
kasuje pamięci: flash, EEPROM ustawiając wartość każdego bitu na 0xff
-F
wyłącza sprawdzanie sygnatury procesora
-t
uruchamia tryb terminala
-V
wyłącza automatyczną weryfikacje zapisu danych

-U ***memtype*:*op*:filename*[:format]***
wskazuje na operację na pamięci. Może to być zapis, odczyt nie tylko pamięci flash, czy EEPROM, ale również fuse bitów procesora
rodzaje *memtype*
eeprom
efuse – rozszerzone fuse bity
flash – pamięć flash procesora
hfuse – high fuse bits
lfuse – low fuse bits
lock – lock bity
Identyfikator ***op*** wskazuje na operacje do wykonania (zapis, odczyt, weryfikacja)
rodzaje *op*
r – read (odczytaj)
w- write (zapisz)
v – verify (sprawdź)
filename wskazuje na plik na którym wykonywana jest operacja. Należy podać nazwę pliku, ewentualnie ścieżkę dostępu do niego.
[:format] wskazuje na format oczekiwanych/przeznaczonych danych. Możliwe identyfikatory:
rodzaje *[:format]*
i – Intel hex
s – motorola S-record
r – raw binary
m – np. przy zapisie fuse bitów, umożliwia podanie wartości w formacie 0x (hex)
a – autodetect

Powyżej przedstawiono podstawowe i najczęściej używane możliwości programu avrdude. Więcej wiadomości zawartych jest w manualu (plik pdf najczęsciej dostarczony z *WinAVR* – katalog bin).

Przykłady użycia programu AVRdude

Przykład nr 1. Zapis do pamięci flash mikrokontrolera *ATmega8* za pomocą programatora *STK200* programu o kodzie wynikowym main.hex

avrdude –p m8 –c stk200 –U flash:w:main.hex

Przykład nr 2. Zapis do pamięci flash uC *ATtiny2313* za pomocą programatora *USBasp* programu o kodzie wynikowym hello.hex, wraz z zapisem fuse bitów.

avrdude –p t2313 –c usbasp –U flash:w:hello.hex –U lfuse:w:0xef:m
–U hfuse:w:0xc9:m

Przykład nr 3. Odczyt z procesora ATmega128 pamięci flash z zapisem do pliku za pomocą programatora *stk500*

avrdude –p m128 –c stk500 –U flash:r:"c:/flash.bin":r

WinAvr – współpraca z AVRdude

Częste korzystanie z AVRdude sprawia iż programowanie staje się banalnie łatwe. Niestety niekiedy wygoda nie jest zbyt duża. Problem polega na ciągłym wpisywaniu w linii komend wywołania programu, albo na ciągłym kopiowaniu pliku wynikowego do katalogu z AVRdude. Na pomoc przychodzi pakiet *WinAVR*, który zawiera kompilatory, debugery i przystosowany jest do pracy z avrgcc. Nie zostaje nic innego aby zintegrować narzędzie ***Programmers Notepad*** wraz z *AVRdude*.

Programmers Notepad zajdziemy w katalogu pn pakietu *WinAVR*. Dostosowanie wygląda następująco. W pliku makefile, plikiem potrzebnym przy kompilacji każdego projektu w C, w którym zapisane są dane o rodzaju procesora, częstotliwości z którą jest taktowany, pliku źródłowym projektu itd., należy umieścić wiadomość o AVRdude. Powinniśmy znaleźć/wpisać tam następujący fragment

AVRDUDE_PROGRAMMER = usbasp
AVRDUDE_WRITE_FLASH = -U flash:w:$(TARGET).hex
AVRDUDE_WRITE_EEPROM = -U eeprom:w:$(TARGET).eep
AVRDUDE_WRITE_LOCK = -U lock:w:**0xff**:m
AVRDUDE_WRITE_HFUSE = -U hfuse:w:**0xC9**:m
AVRDUDE_WRITE_LFUSE = -U lfuse:w:**0x9F**:m

AVRDUDE_FLAGS = -p $(MCU) -P $(AVRDUDE_PORT) -c $(AVRDUDE_PROGRAMMER)

Jeśli któraś z opcji nie jest potrzebna zaleca się jej zakomentowanie.
W pliku makefile powinna znaleźć się również linia:

program: $(TARGET).hex $(TARGET).eep
$(AVRDUDE) $(AVRDUDE_FLAGS) $(AVRDUDE_WRITE_FLASH)
$(AVRDUDE_WRITE_EEPROM) $(AVRDUDE_WRITE_HFUSE) $(AVRDUDE_WRITE_LFUSE)
$(AVRDUDE_WRITE_LOCK)

Tak przygotowany plik makefile, zapewnia obsługe programowania z poziomu Programmers Notepad, poprzez wywołanie odpowiedniego makra z zakładek tools. Jeśli nie zainstalowaliśmy tam patha o automatycznej nazwie ***[WinAvr]program***, możemy to uczynić ręcznie. Należy przejść *Tools→Options→Tools→None(Global Tools)*. Następnie dodać opcje *[WinAvr]program*, poprzez wskazanie ścieżki dostępu do pliku ***program.bat***, którego listing znajduje się poniżej. Jako katalog projektu należy wpisać *$(ProjectPath)*. Jako skrót wywołania można użyć dowolnego klawisza (zalecane F7-F10).

```
@set PATH=C:\WinAvr-20070525\bin;C:\WinAvr-20070525\utils\bin;
make.exe program
```

Jeśli tylko skompilowaliśmy plik źródłowy możemy wywołać programowanie przez odpowiedni klawisz lub z zakładki Tools. Tak oto można skonfigurować *WinAvr* do współpracy z *AVRdude*.

Opis biblioteki AVRlibc

Biblioteka ta jest podstawowym narzędziem pracy przy programowaniu w C. Zawiera podstawowe funkcje i procedury umieszczone w odpowiednich modułach. Poniżej znajduje się dosyć zgrubny opis tychże modułów.

Lista plików nagłówkowych

avr/crc16.h	Obliczanie 16 bitowego CRC
avr/delay.h	Funkcje opóźniające (w rozwoju)
avr/eeprom.h	Funkcje dostępu do wewnętrznej pamięci EEPROM
avr/interrupt.h	Funkcje obsługi przerwań
avr/io.h	Włącza pozostałe nagłówki I/O
avr/io[MCU].h	Definicje I/O dla różnych mikrokontrolerów AVR
avr/parity.h	Obliczanie bitu parzystości
avr/pgmspace.h	Funkcje dostępu do pamięci programu
avr/sfr_defs.h	Makra dla peryferii
avr/signal.h	Obsługa przerwań i sygnałów AVR
avr/sleep.h	Zarządzanie poborem energii
avr/timer.h	Funkcje dla licznika/czasomierza 0
avr/twi.h	Obsługa TWI (i2c) w ATMega
avr/wdt.h	Funkcje kontrolujące układ watchdoga
ctype.h	Funkcje testujące wartości typów znakowych
errno.h	Obsługa błędów
inttypes.h	Definicje różnych typów całkowitych
math.h	Różne funkcje matematyczne
setjmp.h	Zawiera funkcje długich skoków (long jumps)
stdio.h	Standardowa biblioteka wejścia/wyjscia
stdlib.h	Rozmaite funkcje standardowe
string.h	Funkcje operujące na łańcuchach

Opis poszczególnych modułów

avr/crc16.h

Zawiera funkcje obliczające 16 bitowe CRC

unsigned int _crc16_update(unsigned int _crc, unsigned char _data)

Oblicza 16 bitowe CRC według standardu *CRC16* (*x^16 + x^15 + x^2 + 1*).

avr/delay.h

Zawiera proste funkcje wstrzymujące działanie programu na pewien czas

void _delay_loop_1(unsigned char _count)

8 bitowy licznik, 3 cykle procesora na _count

void _delay_loop_2(unsigned int _count)

16 bitowy licznik, 4 cykle procesora na _count

void _delay_ms (double _ms)
Wstrzymuje działanie programu na _ms millisekund, używając _delay_loop_2().
Makro *F_CPU* powinno zawiarac częstotliwość zegara w hercach.
Maksymalne możliwe wstrzymanie to 262.14 ms / (*F_CPU w MHz*)

void _delay_us (double __us)
Wstrzymuje działanie programu na __us mikrosekund, używając _delay_loop_1().
Makro *F_CPU* powinno zawiarac częstotliwość zegara w hercach.
Maksymalne możliwe wstrzymanie to 768 us / (*F_CPU w MHz*)

avr/eeprom.h

Zawiera funkcje dostępu do wewnętrznej pamięci *EEPROM*

int eeprom_is_ready()
Zwraca wartość różną od 0 jeżeli *EEPROM* jest gotowy na następną operację (bit *EEWE* w rejestrze *EECR* jest równy 0)

unsigned char eeprom_read_byte(unsigned int *addr)
Czyta jeden bajt z *EEPROMu* spod adresu *addr*.

unsigned int eeprom_read_word(unsigned int *addr)
Czyta 16-bitowe słowo z EEPROMu spod adresu *addr*.

void eeprom_write_byte(unsigned int *addr, unsigned char val);
Zapisuje bajt *val* do *EEPROMu* pod adres *addr*.

void eeprom_read_block(void *buf, unsigned int *addr, size_t n);
Czyta blok o wielkości *n* bajtów z *EEPROMu* spod adresu *addr* do *buf*.

avr/interrupt.h

Zawiera funkcje obsługi przerwań.

sei()
Włącza przerwania. Makro.

cli()
Wyłącza przerwania. Makro.

void enable_external_int(unsigned char ints)
Wpisuje ints do rejestrów EIMSK lub GIMSK, w zależności, który rejestr zdefiniowany w mikrokontrolerze: EIMSK lub GIMSK.

void timer_enable_int(unsigned char ints);
Wpisuje *ints* do rejestru TIMSK, jeżeli TIMSK jest zdefiniowany

avr/io.h

Włącza pliki nagłówkowe *avr/sfr_defs.h* oraz odpowiedni *avr/io[MCU].h*. Służy do definiowania stałych specyficznych dla danego mikrokontrolera na podstawie parametru -*mmcu=typ_MCU* przekazanego do kompilatora. Ten plik powinien być włączany w każdym programie na mikrokontroler AVR.

avr/io[MCU].h

Definicje rejestrów I/O dla odpowiedniego typu mikrokontrolera, gdzie [MCU] jest tekstem określającym typ w rodzaju *2313*, *8515* itp. Zobacz do dokumentacji mikrokontrolera. Tych plików nie należy włączać do pisanych programów – robi to za nas *avr/io.h* na podstawie parametru -*mmcu=typ_MCU* przekazanego do kompilatora np. w pliku *makefile*.
avr/parity.h

Zawiera definicje funkcji pomocnej w obliczaniu bitu parzystości lub nieparzystości.
parity_even_bit(val)

avr/pgmspace.h

Zawiera funkcje dostępu do danych znajdujących się w pamięci programu.

#define PGM_P const prog_char *
Służy do deklaracji zmiennej, która jest wskaźnikiem do łańcucha znaków w pamięci programu.

#define PGM_VOID_P const prog_void *
Służy do deklaracji wskaźnika do dowolnego obiektu w pamięci programu.

#define PSTR(s) ({static char __c[] PROGMEM = (s); __c;})
Służy do deklaracji wskaźnika do łańcuha znaków w pamięci programu.

unsigned char __elpm_inline(unsigned long __addr) [static]
Służy do odczytania zawartości pamięci programu o adresie powyżej 64kB (ATmega103, ATmega128). Jeżeli jest to możliwe, należy umieścić tablice ze stałymi poniżej granicy 64kB (jest to bardziej efektywne rozwiązanie).

void *memcpy_P(void* dest, PGM_VOID_P src, size_t n)
Kopiuje *n* znaków z jednego ciągu do drugiego. Jako wynik zwraca wskaźnik do *dest*. Jest odpowiednikiem funkcji *memcpy()* z tą różnicą, że łańcuch *src* znajduje się w pamięci programu.

int strcasecmp_P(const char* s1, PGM_P s2)
Porównuje *s1* z *s2*, ignorując wielkość liter. Parametr *s1* jest wskaźnikiem do łańcucha znajdującego się w pamięci SRAM. Parametr *s2* jest wskaźnikiem do łańcucha znajdującego się w pamięci programu. Zwraca wartość mniejszą od 0 jeżeli *s1* jest mniejsze od *s2*. Zero jeśli są równe. Większą od zera jeśli *s1* jest większe od *s2*. Jest odpowiednikiem funkcji *strcasecmp()* z tą różnicą, że łańcuch *s2* znajduje się w pamięci programu.

char *strcat_P(char* dest, PGM_P src)
Dołącza znaki jednego ciągu do drugiego. Jako wynik zwraca wskaźnik do *dest*. Jest odpowiednikiem funkcji *strcat()* z tą różnicą, że łańcuch *src* znajduje się w pamięci programu.

int strcmp_P(const char* s1, PGM_P s2)
Porównuje *s1* z *s2*, uwzględniając wielkość liter. Parametr *s1* jest wskaźnikiem do łańcucha znajdującego się w pamięci SRAM. Parametr *s2* jest wskaźnikiem do łańcucha znajdującego się w pamięci programu. Zwraca wartość mniejszą od 0 jeżeli *s1* jest mniejsze od *s2*. Zero jeśli są równe. Większą od zera jeśli *s1* jest większe od *s2*. Jest odpowiednikiem funkcji *strcmp()* z tą różnicą, że łańcuch *s2* znajduje się w pamięci programu.

char* strcpy_P(char* dest, PGM_P src)
Kopiuje *src* do *dest*. Jako wynik zwraca wskaźnik do *dest*. Jest odpowiednikiem funkcji *strcpy()* z tą różnicą, że łańcuch *src* znajduje się w pamięci programu.

size_t strlen_P(PGM_P src)
Zwraca ilość znaków w *src*. Jest odpowiednikiem funkcji *strlen()* z tą różnicą, że łańcuch *src* znajduje się w pamięci programu.

int strncasecmp_P(const char *s1, PGM_P s2, size_t n)
Porównuje pierwszych *n* znaków *s1* z *s2*, ignorując wielkość liter. Parametr *s1* jest wskaźnikiem do łańcucha znajdującego się w pamięci *SRAM*. Parametr *s2* jest wskaźnikiem do łańcucha znajdującego się w pamięci programu. Parametr *n* określa ile znaków ma być porównywanych. Zwraca wartość mniejszą od 0 jeżeli pierwsze *n* znaków *s1* jest mniejsze od *s2*. Zero jeśli są równe. Większą od zera jeśli *s1* jest większe od *s2*. Jest odpowiednikiem funkcji *strncasecmp()* z tą różnicą, że łańcuch *s2* znajduje się w pamięci programu.

int strncmp_P(const char* s1, PGM_P s2, size_t n)
Porównuje pierwszych *n* znaków *s1* z *s2*, uwzględniając wielkość liter. Parametr *s1* jest wskaźnikiem do łańcucha znajdującego się w pamięci *SRAM*. Parametr *s2* jest wskaźnikiem do łańcucha znajdującego się w pamięci programu. Parametr *n* określa ile znaków ma być porównywanych. Zwraca wartość mniejszą od 0 jeżeli pierwsze n znaków *s1* jest mniejsze od *s2*. Zero jeśli są równe. Większą od zera jeśli *s1* jest większe od *s2*. Jest odpowiednikiem funkcji *strncmp()* z tą różnicą, że łańcuch *s2* znajduje się w pamięci programu.

char* strncpy_P(char* dest, PGM_P src, size_t n)
Kopiuje nie więcej niż *n* bajtów z *src* do *dest*. Jako wynik zwraca wskaźnik do *dest*. Jest odpowiednikiem funkcji *strncpy()* z tą różnicą, że łańcuch *src* znajduje się w pamięci programu.

avr/sfr_defs.h

Zawiera wiele bardzo przydatnych makr dla dostępu do portów wejścia/wyjścia.

_BV(x)
Zwraca wartość bitu (bit value) *x*. Zdefiniowany jako (1 << x). Makro.

inb(sfr)
Czyta bajt z *sfr*. Makro.

outb(sfr, val)
Wpisuje *val* do sfr. Makro. Odwrotnie jak inb(sfr).

cbi(sfr, bit)
Kasuje *bit* w *sfr*. Makro.

sbi(sfr, bit)
Ustawia *bit* w *sfr*. Makro.

bit_is_set(sfr, bit)
Zwraca wartość różną od 0, jeżeli *bit* w *sfr* jest ustawiony, w przeciwnym wypadku 0. Makro.

bit_is_clear(sfr, bit)
Zwraca wartość różną od 0, jeżeli *bit* w *sfr* jest skasowany, w przeciwnym wypadku 0. Makro.

loop_until_bit_ist_set(sfr, bit)
Wstrzymuje działanie programu (wykonuje pętlę) dopóki *bit* w *sfr* jest ustawiony. Makro.

loop_until_bit_is_clear(sfr, bit)
Wstrzymuje działanie programu (wykonuje pętlę) dopóki *bit* w *sfr* jest skasowany. Makro.

avr/signal.h

Definiuje nazwy uchwytów dla przerwań, które znajdują się na początku pamięci FLASH. Oto one:

SIG_INTERRUPT0 do SIG_INTERRUPT7
Uchwyty funkcji obsługi przerwań zewnętrznych od 0 do 7. Przerwania o numerach większych od 1 są dostępne tylko w niektórych układach ATmega.

SIG_OUTPUT_COMPARE2
Uchwyt funkcji obsługi przerwania od porównania licznika 2.

SIG_OVERFLOW2
Uchwyt funkcji obsługi przerwania do przepełnienia licznika 2.

SIG_INPUT_CAPTURE1
Uchwyt funkcji obsługi przerwania od przechwytywania licznika 1.

SIG_OUTPUT_COMPARE1A
Uchwyt funkcji obsługi przerwania od porównania licznika 1 (A).

SIG_OUTPUT_COMPARE1B
Uchwyt funkcji obsługi przerwania od porównania licznika 1 (B).

SIG_OVERFLOW1
Uchwyt funkcji obsługi przerwania do przepełnienia licznika 1.

SIG_OUTPUT_COMPARE0
Uchwyt funkcji obsługi przerwania od porównania licznika 0.

SIG_OVERFLOW0
Uchwyt funkcji obsługi przerwania do przepełnienia licznika 0.

SIG_SPI
Uchwyt funkcji obsługi przerwania SPI.

SIG_UART_RECV
Uchwyt funkcji obsługi przerwania UART(0) – odbiór znaku.

SIG_UART1_RECV
Uchwyt funkcji obsługi przerwania UART1 – odbiór znaku. UART1 jest dostępny w niektórych układach ATmega.

SIG_UART_DATA
Uchwyt funkcji obsługi przerwania UART(0) – pusty rejestr danych.

SIG_UART1_DATA
Uchwyt funkcji obsługi przerwania UART1 – pusty rejestr danych. UART1 jest dostępny tylko w niektórych układach ATmega.

SIG_UART_TRANS
Uchwyt funkcji obsługi przerwania UART(0) – zakończenie transmisji.

SIG_UART1_TRANS
Uchwyt funkcji obsługi przerwania UART1 – zakończenie transmisji. UART1 jest dostępny tylko w niektórych układach ATmega.

SIG_ADC
Uchwyt funkcji obsługi przerwania ADC – zakończenie przetwarzania.

SIG_EEPROM
Uchwyt funkcji obsługi przerwania EEPROM – gotowość.

SIG_COMPARATOR
Uchwyt funkcji obsługi przerwania z komparatora analogowego.

SIGNAL(signame)
Używany do definicji uchwytu sygnału dla signame.

INTERRUPT(signame)
Używany do definicji uchwytu przerwania dla signame.
Dla uchwytu zdefiniowanego w *SIGNAL()*, dodatkowe przerwania są bezwarunkowo zabronione, natomiast w uchwycie *INTERRUPT()*, pierwszą (bezwarunkowo) instrukcją jest sei, i występujące w tym czasie przerwania mogą być obsługiwane.

avr/sleep.h

Zawiera definicje i funkcje pomocne w zarządzaniu poborem energii.

#define SLEEP_MODE_ADC
Redukcja zakłóceń z przetwornika analogowo/cyfrowego.

#define SLEEP_MODE_EXT_STANDBY
Rozszerzony tryb gotowości (Extended Standby).

#define SLEEP_MODE_IDLE
Tryb bezczynny (Idle).

#define SLEEP_MODE_PWR_DOWN
Wyłączenie zasilania (Power Down).

#define SLEEP_MODE_PWR_SAVE
Oszczędzanie zasilania (Power Save).
#define SLEEP_MODE_STANDBY
Tryb gotowości (Standby).
void set_sleep_mode(uint8_t mode)
Ustawia bity w rejestrze MCUCR aby wybrać odpowiedni tryb uśpienia.
void sleep_mode(void)
Wprowadza kontroler w tryb uśpienia na podstawie wcześniej wybranego trybu za pomocą funkcji set_sleep_mode(). Aby uzyskać więcej informacji, zobacz do dokumentacji mikrokontrolera.

avr/timer.h
Zawiera definicje funkcji kontrolujących działanie licznika/czasomierza 0.

void timer0_source(unsigned int src)
Wpisuje src w rejestr TCCR0. Wartość src może przyjmować następujące wartości symboliczne:

```
enum {
STOP = 0,
CK = 1,
CK8 = 2,
CK64 = 3,
CK256 = 4,
CK1024 = 5,
T0_FALLING_EDGE = 6,
T0_RISING_EDGE = 7
};
```

void timer0_stop()
Zatrzymuje Timer 0 poprzez wyzerowanie rejestru *TCNT0*.
void timer0_start()
Startuje Timer 0 poprzez wpisanie 1 w rejestr *TCNT0*.

avr/twi.h
Definiuje kilka stałych dla obsługa magistrali *TWI (i2c)* w *ATMega*.

avr/wdt.h
Zawiera definicje i funkcje pomocne w używaniu układu watchdoga.

wdt_reset()
Powoduje kasowanie czasomierza układu Watchdog.
wdt_enable(timeout)
Ustawia odpowiedni timeout i uruchamia układ watchdoga. Zobacz do dokumentacji Atmel AVR. Wartość timeout może przyjmować jedną z predefiniowanych wartości:

```
WDTO_15MS
WDTO_30MS
WDTO_60MS
WDTO_250MS
WDTO_500MS
WDTO_1S
WDTO_2S
```

wdt_disable()
Wyłącza układ watchdoga.

ctype.h

Zawiera definicje funkcji testujących i zamieniających typy znakowe.

int isalnum(int __c);
Zwraca 1 jeżeli __c jest cyfrą lub literą, w przeciwnym wypadku 0.
int isalpha(int __c);
Zwraca 1 jeżeli __c jest literą, w przeciwnym wypadku 0.
int isascii(int __c);
Zwraca 1 jeżeli __c zawiera się w 7 bitowym ASCII, w przeciwnym wypadku 0.
int iscntrl(int __c);
Zwraca 1 jeżeli __c jest znakiem kontrolnym, w przeciwnym wypadku 0.
int isdigit(int __c);
Zwraca 1 jeżeli __c jest cyfrą, w przeciwnym wypadku 0.
int isgraph(int __c);
Zwraca 1 jeżeli __c jest „drukowalne” (z wyjątkiem spacji), w przeciwnym wypadku 0.
int islower(int __c);
Zwraca 1 jeżeli __c jest małą literą alfabetu, w przeciwnym wypadku 0.
int isprint(int __c);
Zwraca 1 jeżeli __c jest „drukowalne” (ze spacją), w przeciwnym wypadku 0.
int ispunct(int __c);
Zwraca 1 jeżeli __c jest znakiem interpunkcyjnym, w przeciwnym wypadku 0.
int isspace(int __c);
Zwraca 1 jeżeli __c jest spacją lub '\n', '\f', '\r', '\t', '\v', w przeciwnym wypadku 0.
int isupper(int __c);
Zwraca 1 jeżeli __c jest dużym znakiem alfanumerycznym, w przeciwnym wypadku 0.
int isxdigit(int __c);
Zwraca 1 jeżeli __c jest cyfrą szesnastkową (0-9 lub A-F), w przeciwnym wypadku 0.
int toascii(int __c);
Zamienia __c na 7 bitowy znak ASCII.
int tolower(int __c);
Zamienia __c na małą literę.
int toupper(int __c);
Zamienia __c na dużą literę.

errno.h

Obsługa błędów.

int errno;
Przechowuje systemowy kod błędu

inttypes.h

Definiuje typy danych całkowitych.

typedef signed char int8_t;
typedef unsigned char uint8_t;

Typy 8-bitowe

typedef int int16_t;
typedef unsigned int uint16_t;

Typy 16-bitowe

typedef long int32_t;
typedef unsigned long uint32_t;

Typy 32-bitowe

typedef long long int64_t;
typedef unsigned long long uint64_t;

Typy 64-bitowe

typedef int16_t intptr_t;
typedef uint16_t uintptr_t;

Typy wskaźnikowe
Należy świadomie używać opcji kompilatora -mint8 – nie będą wtedy dostępne typy 32 i 64 bitowe.

math.h

M_PI = 3.14159265358979323846264
Liczba PI.
M_SQRT2 = 1.41421356237309504880168872
Pierw. kwadr. z 2
double cos(double x)
Zwraca cosinus z x.
double fabs(double x)
Zwraca absolutną wartość z x.
double fmod(double x, double y)
Zwraca zmiennoprzecinkową resztę z dzielenia x/y.
double modf(double x, double *iptr)
Zwraca część ułamkową z x i zapamiętuje część całkowitą w *iptr.
double sin(double x)
Zwraca sinus z x.
double sqrt(double x)
Zwraca pierwiastek kwadratowy z x.
double tan(double x)
Zwraca tangens z x.
double floor(double x)
Zwraca większą wartość całkowitą mniejszą niż x.
double ceil(double x)
Zwraca mniejszą wartość całkowitą większą niż x.
double frexp(double x, int *exp)
Rozdziela x na znormalizowany ułamek, który jest zwracany, i na wykładnik, który jest zapamiętany w *exp.
double ldexp(double x, int exp);
Zwraca x^exp.

double exp(double x)
Zwraca e^x.
double cosh(double x)
Zwraca cosinus hiperboliczny z x.
double sinh(double x)
Zwraca sinus hiperboliczny z x.
double tanh(double x)
Zwraca tangens hiperboliczny z x.
double acos(double x)
Zwraca arcus cosinus z x.
double asin(double x)
Zwraca arcus sinus z x.
double atan(double x)
Zwraca arcus tangens z x. Wyjście między -PI/2 i PI/2 (włącznie).
double atan2(double x, double y)
Zwraca arcus tangens z x/y. Zwraca uwagę na znak argumentów. Wyjście pomiędzy -PI a PI (włącznie).
double log(double x)
Zwraca logarytm naturalny z x.
double log10(double x)
Zwraca logarytm dziesiętny z x.
double pow(double x, double y)
Zwraca x^y.
double strtod(const char *s, char **endptr)
Zamienia łańcuch ASCII na liczbę typu double.
double square(double x)
Zwraca x2.
double inverse(double x)
Zwraca 1/x.

UWAGA. *Aby skorzystać z tych funkcji należy włączyć do projektu bibliotekę libm.a.*

setjmp.h

int setjmp(jmp_buf env)
Deklaruje długi skok do miejsca przeznaczenia wykonanego przez *longjmp()*.
void longjmp(jmp_buf env, int val)
Wykonuje długi skok do pozycji wcześniej zdefiniowanej przez *setjmp(env)*, która powinna zwrócić val.

stdlib.h

Definiuje następujące typy:

```
typedef struct {
int quot;
int rem;
} div_t;

typedef struct {
long quot;
long rem;
} ldiv_t;
```

typedef int (*__compar_fn_t)(const void *, const void *);
Używane w funkcjach porównujących np. qsort().

void abort();
Skutecznie przerywa wykonywanie programu przez wprowadzenie MCU w nieskończoną pętlę.

long labs(long x);
Zwraca absolutną wartość x typu long.

div_t div(int x, int y);
Dzieli x przez y i zwraca rezultat (iloraz i resztę) w strukturze div_t.

ldiv_t ldiv(long x, long y);
Dzieli x przez y i zwraca rezultat (iloraz i resztę) w strukturze ldiv_t.

void qsort(void *base, size_t nmemb, size_t size, __compar_fn_t compar);
Sortuje tablicę base z nmemb elementami rozmiaru size, używając funkcji porównującej compar.

long strtol(const char *nptr, char **endptr, int base);
Zamienia łańcuch nptr według podstawy base na liczbę typu long.

unsigned long strtoul(const char *nptr, char **endptr, int base);
Zamienia łańcuch nptr według podstawy base na liczbę typu unsigned long.

long atol(char *p);
Zamienia łańcuch p na liczbę typu long.

int atoi(char *p);
Zamienia łańcuch p na liczbę typu int.

void *malloc(size_t size);
Alokuje size bajtów pamięci i zwraca wskaźnik do niego.

void free(void *ptr);
Zwalnia pamięć wskazywaną przez ptr, która była wcześniej zaalokowana funkcją malloc().

char *itoa(int value, char *string, int radix);
Zamienia liczbę całkowitą na łańcuch. Nie jest kompatybilna z ANSI C, lecz może być użyteczna.

<u>string.h</u>

void *memcpy(void *to, void *from, size_t n);
Kopiuje n bajtów z from do to.

void *memmove(void *to, void *from, size_t n);
Kopiuje n bajtów z from do to, gwarantując poprawność zachowania dla nakładających się łańcuchów.

void *memset(void *s, int c, size_t n);
Ustawia n bajtów z s na wartość c.

int memcmp(const void *s1, const void *s2, size_t n);
Porównuje n bajtów między s1 a s2.

void *memchr(void *s, char c, size_t n);
Zwraca wskaźnik do pierwszego wystąpienia c w pierwszych n bajtach s.

size_t strlen(char *s);
Zwraca długość łańcucha s.

char *strcpy(char *dest, char *src);
Kopiuje src do dest. Jako wynik zwraca wskaźnik do dest.

char *strncpy(char *dest, char *src, size_t n);
Kopiuje nie więcej niż n bajtów z src do dest. Jako wynik zwraca wskaźnik do dest.

char *strcat(char *dest, char *src);
Dołącza src do dest. Jako wynik zwraca wskaźnik do dest.

char *strncat(char *dest, char *src, size_t n);
Dołącza nie więcej niż n bajtów z src do dest. Jako wynik zwraca wskaźnik do dest.

int strcmp(const char *s1, const char *s2);
Porównuje s1 z s2, uwzględniając wielkość liter. Zwraca wartość mniejszą od 0 jeżeli s1 jest mniejsze od s2. Zero jeśli są równe. Większą od zera jeśli s1 jest większe od s2.

int strncmp(const char *s1, const char* s2, size_t n);
Porównuje pierwszych n znaków s1 z s2, uwzględniając wielkość liter. Parametr n określa ile znaków ma być porównywanych. Zwraca wartość mniejszą od 0 jeżeli pierwsze n znaków s1 jest mniejsze od s2. Zero jeśli są równe. Większą od zera jeśli s1 jest większe od s2.

strdupa(s);
Duplikuje s, zwracając identyczny łańcuch. Makro.

strndupa(s, n);
Zwraca zaalokowaną kopię n batów z s. Makro.

char *strchr(const char *s, int c);
Zwraca wskaźnik do pierwszego wystąpienia c w s.

char *strrchr(const char *s, int c);
Zwraca wskaźnik do ostatniego wystąpienia c w s.

size_t strnlen(const char *s, size_t maxlen);
Zwraca długość łańcucha s, ale nie więcej niż maxlen.

void *memccpy(void *dest, const void *src, int c, size_t n);
Kopiuje nie więcej niż n bajtów z src do dest dopóki zostanie znaleziony c.

int strcasecmp(const char *s1, const char *s2);
Porównuje s1 z s2, ignorując wielkość liter. Zwraca wartość mniejszą od 0 jeżeli s1 jest mniejsze od s2. Zero jeśli są równe. Większą od zera jeśli s1 jest większe od s2.

char *strlwr(char *s);
Zamienia wszystkie duże litery w łańcuchu s na małe.

int strncasecmp(const char *s1, const char *s2, size_t n);
Porównuje n bajtów z s1 i s2, ignorując wielkość znaków.

char *strrev(char *s1);
Odwraca kolejność znaków w s1.

char *strstr(const char *haystack, const char *needle);
Znajduje needle w haystack, i zwraca wskaźnik do niego.

char *strupr(char *s);
Zamienia wszystkie małe litery w łańcuchu s na duże.

Bardzo szczegółowy opis zawierający wszystkie funkcje znajduje się w manualu biblioteki *avr-libc*. Dołączony jest on do pakietu *WinAVR*.

PROGRAMOWANIE W JĘZYKU C

Mikrokontrolery *AVR* ostatnimi laty stały się bardzo popularne. Popularność ich zaowocowała różnymi sposobami tworzenia dla nich oprogramowania. Producent *AVR* firma *Atmel* wspomaga programowanie procesorów swoim pakietem *AVRstudio*, umożliwiającym pisanie w asemblerze, a także C. Powstał nawet specjalny dialekt *Basica*, mianowicie *Bascom*. Ten specyficzny język wspierany jest przez pakiet *Bascom-AVR*.

Jak zwykle w sytuacjach, gdzie mamy więcej niż jeden sposób dotarcia do celu powstał konflikt. Konflikt ten opiera się na pytaniu: „Czy lepszy Bascom, czy C, a może asembler". Na forach internetowych mnoży się od takich wątków, tysiąców postów. Prawda jest taka, że nie da się ocenić który język jest lepszy. Autor nawet nie będzie się starał przekonywać do programowania w C, bo sam zaczynał od Bascoma i mile wspomina ten czas. Poda tylko zalety języka C.

Dlaczego programować w C? Przyczyn jest wiele. Główna to darmowe kompilatory, edytury i debugery oraz wszechobecna literatura na temat programowania w tym języku. Znajomość C przy programowaniu uC, może przydać się w projektach wyłącznie informatycznych, bez mikroprocesorów.

Aby nauczyć się programować w C potrzebna jest następująca wiedza. Znajomość składni C, podstawowych funkcji (dodawanie, odejmowanie itd.), działań bitowych, używania modułów itd. W tym opracowaniu znajdują się zagadnienia, które zdaniem autora będą bardzo przydatne.

Operatory bitowe w C

Operatory bitowe pozwalają nam na operacje na bitach typów całkowitych takich jak *char, int, long int*. W języku c mamy do dyspozycji 6 takich operatorów.

~ - bitowa negacja
| - bitowa alternatywa (OR)
& - bitowa koniunkcja (AND)
^ - bitowa różnica symetryczna (XOR)
>> - bitowe przesunięcie w prawo
<< - bitowe przesunięcie w lewo

Żeby dobrze zrozumieć operacje na bitach należy dobrze poznać sposób przechowywania tych liczb w pamięci komputera. Liczby całkowite przechowywane są w **kodzie U2.**

Zapis liczb w komputerze

Kod U2 jest to kod w którym najstarszy bit ma wagę -2^n a pozostałe $2^{n-1}, \ldots, 2^1, 2^0$. Przy liczbach dodatnich kod ten wygląda jak naturalny kod dwójkowy

Np.
liczba 67 zapisana w kodzie U2 na 8 bitach wygląda : ***01000011***
A liczba -67 wygląda: ***10111101***

Jeżeli mamy liczbę dodatnia i chcemy zrobić z niej liczbę ujemna zapisana w kodzie U2 to negujemy wszystkie bity tej liczby a następnie dodajemy jeden najmłodszy bit.

Np. *44 =* ***00101100***
Negujemy wszystkie bity 11010011

11010011
A następnie dodajemy jeden *+ 00000001*
= 11010100 = -44

Należy jednak pamiętać aby nie przekroczyć zakresu tego kodu. Zakres tego kodu zapisanego na n bitach wynosi $(-2^n, 2^n - 1)$ liczb ujemnych mamy o jedna więcej. Jeżeli będziemy operowali na ośmiu bitach i dodamy

127 = 01111111
+ 1 = 00000001
10000000 = -128

-128 = 10000000
+(-1) = 11111111
1 01111111 = 127

Po przekroczeniu zakresu otrzymaliśmy złą odpowiedź. Przy dodawaniu do siebie dwóch liczb w kodzie U2 bit przepełnienia zostaje pominięty. Kiedy już wiemy jak zapisywane są liczby w komputerze, to możemy teraz przystąpić do omówienia poszczególnych operatorów.

~ *negacja bitowa (NOT)*

Negacja bitowa jest operatorem jednoargumentowym. Operator ten neguje wszystkie bity liczby, przed którą został on umieszczony.

Np. $\sim (47)_{10} = \sim (00101111)_{U2} = (11010000)_{U2} = (-48)_{10}$

Jeżeli zależy nam na szybkim obliczeni wartości negacji danej liczby pomożemy zauważyć, że w wyniku negacji otrzymaliśmy po prostu :
~a = - (a+1)

| *bitowa alternatywa (OR)*

Bitowa alternatywa jest operatorem dwuargumentowym. Zapis ***f = a|b*** oznacza że w liczbie f jedynki ustawiane są na tych pozycjach, na których znajdowały się jedynki w liczbach a i b.

Np.
$a = 7|4 = (00000111)_{U2}|(00000100)_{U2} = (00000111)_{U2} = 7$
$b = 100|57 = (01100100)_{U2}|(00111001)_{U2} = (01111101)_{U2} = 125$
$c = (-25)|55 = (11100111)_{U2}|(00110111)_{U2} = (11110111)_{U2} = -9$

& bitowa koniunkcja (AND)

Bitowa koniunkcja to operator dwuargumentowy. Zapis ***f = a&b*** oznacza że bity w liczbie f są jedynkami, o ile oba bity na tych samych pozycjach w liczbach a i b są jedynkami.

Np.
a = 7&4 = $(00000111)_{U2}$&$(00000100)_{U2}$ = $(00000100)_{U2}$ = 4
b = 100&57 = $(01100100)_{U2}$&$(00111001)_{U2}$= $(00100000)_{U2}$ = 32
c = (-25)&55 = $(11100111)_{U2}$&$(00110111)_{U2}$ = $(00100111)_{U2}$ = 39

Należy pamiętać aby nie pomylić operatorów bitowych ***(~ , | , &)*** z operatorami ***(! , || , &&)*** , które wyglądają dość podobnie jednak robią zupełnie coś innego.

Np.
!(-6) = 0 !0 = 1 ~(-6) = 5 ~0 = -1 7||0 = 1 10||(-2) = 1 7|0 = 1 10|(-2) = -1 7&&(-1) = 1 7&&8 = 1 7&(-1) = 7 7&8 = 0

^ bitowa różnica symetrzyczna (XOR)

Bitowa różnica symetryczna jest operatorem dwuargumentowym. Zapis ***f = a^b*** oznacza że bity w liczbie f będą równe jeden jeżeli bity na tych samych pozycjach w liczbach a i b są różne. Jeżeli są sobie równe to odpowiadający im bit w liczbie f będzie równy 0.

Np.
a = 90^(-24) =$(01011010)_{U2}$^$(11101000)_{U2}$ = $(10110010)_{U2}$ = -78
b = 127^85 =$(01111111)_{U2}$^$(01010101)_{U2}$ = $(00101010)_{U2}$ = 42
c =(-5)^(-44) =$(11111011)_{U2}$^$(11010100)_{U2}$ = $(00101111)_{U2}$ = 47

>> bitowe przesunięcie w prawo

Zapis ***f = a>>n*** oznacza że z liczby a zabieramy n bitów przesuwając wszystkie bity w prawo. Przy przesunięciu o jeden bit wszystkie bity są przesuwane o jedną pozycję w prawo a najstarszy bit zachowuje swoją dotychczasowa wartość. Operacja ta odpowiada podzieleniu liczby przez 2^n i zaokrągleniu wyniku w dół. Ma to znaczenie w liczbach ujemnych.

Np.
a = 80>>2 = $(01010000)_{U2}$>>2 = $(00010100)_{U2}$ = 20
b =127>>5 = $(01111111)_{U2}$>>5 = $(00000011)_{U2}$ = 3
c = -15>>2 = $(11110001)_{U2}$>>2 = $(11111100)_{U2}$ = -4
d = -15/4 = -3.

<< przesunięcie bitowe w lewo

Zapis ***f = a<<n*** oznacza że do liczby a dodajemy n bitów przesuwając wszystkie bity w lewo. Podczas przesuwania na miejsca najmłodszych bitów wstawiane są 0 . Operacja ta odpowiada szybkiemu pomnożeniu liczby przez 2^n. Operacje tę możemy wykorzystać np. w algorytmie Karatsuby.

Np.
a = 5<<4 = $(00000101)_{U2}$<<4 = $(01010000)_{U2}$ = 80

b =(-7)<<3 = $(11111001)_{U2}$<<3 = $(11001000)_{U2}$ = -56
c = (-1)<<7 = $(11111111)_{U2}$<<7 = $(10000000)_{U2}$ = -128

Priorytety operacji

W operacjach na bitach istotną rolę odgrywają priorytety operacji.
Operatory Łączność prawostronna lewostronna lewostronna lewostronna lewostronna lewostronna lewostronna lewostronnalewostronna lewostronna lewostronna prawostronna prawostronna lewostronna

Operatory	Łączność
() [] ->	*lewostronna*
*! ~ ++ -- + - * &*	*prawostronna*
** / %*	*lewostronna*
+ -	*lewostronna*
<< >>	*lewostronna*
< <= > >=	*lewostronna*
== !=	*lewostronna*
&	*lewostronna*
^	*lewostronna*
|	*lewostronna*
&&	*lewostronna*
||	*lewostronna*
?:	*prawostronna*
*= += -= *= /= %= ^= |= <<= >>=*	*prawostronna*
,	*lewostronna*

Łączność lewostronna oznacza , że jeżeli w wyrażeniu znajdują co najmniej dwa operatory na tym samym stopniu struktury nawiasowej to najpierw wykonywany jest operator lewy.
W tej tablicy dwa razy występują operatory **(+,-,*,&)** ponieważ operatory jednoargumentowe mają większy priorytet niż operatory dwuargumentowe. No i tak np. znak – jako operator jednoargumentowy przed zmienną zmienia jej znak, zaś operator & jako operator jednoargumentowy służy do wyłuskania adresu zmiennej. Operatory bitowe możemy np. wykorzystać do wyciągnięcia jednego bitu bądź kilku bitów z jakiejś liczby. Wykorzystując wyciąganie jednego bitu można stworzyć np. typ logiczny 0-fałsz 1- prawda.

Wprowadzenie do programowania w języku C

W tej części opracowania zostały zawarte najprostsze programy wraz z ich opisami.

Najprostszy program w języku C wygląda następująco:

```
 main()
{
}
```

Składa się on z procedury *main()* , która nic nie robi. Zwróć uwagę na sposób zapisu tej procedury. Stosujemy tutaj nawiasy zwykłe i klamrowe. Zawsze musisz je stosować tak jak w powyższym przykładzie. Podany powyżej przykład najprostszego programu w języku C pokazuje jedynie ogólną ideę zapisu. W praktyce, kompilator AVR-GCC wymaga nieco więcej informacji o pisanej procedurze. Dlatego w naszym prawdziwym programie funkcja *main()* wygląda następująco:

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

Jest to FUNKCJA *main()*, a działanie funkcji polega na wykonaniu pewnych działań i zwróceniu wyniku tych działań. W odróżnieniu od funkcji , PROCEDURA wykonuje jakieś czynności , ale nie zwraca ich wyniku. Tak więc wiesz już , że *main()* to funkcja, bo zwraca wynik swoich działań. Słowo *int* poprzedzające nawę funkcji *main()*, informuje kompilator jakiego typu dane zostaną zwrócone przez funkcję *main()*. W tym przypadku chodzi o liczbę całkowitą , ze znakiem – ten typ danych oznacza się właśnie jako **int**.

Jak widać zwykły nawias znajdujący się po nazwie procedury *main()* nie jest wcale pusty a zawiera tajemnicze słowo **void**. Miejsce pomiędzy nawiasami zwykłymi jest przeznaczone na podanie informacji o argumentach przekazywanych do funkcji lub procedury w momencie gdy będzie ona wywoływana do zastosowania. Najprostszym argumentem funkcji jest właśnie **void** ,czyli NIC.

Słowo *void* informuje kompilator, że do funkcji *main()* nie będzie przekazywany w momencie jej wywoływania żaden argument.

W drugiej linii widzimy otwarcie nawiasu klamrowego **{** , a w piątej linii jego zamknięcie **}**. Nawias klamrowy obejmuje całą zawartość funkcji lub procedury. Zawartość tą fachowcy nazywają „ciałem" funkcji lub procedury. W praktyce, pisząc programy , należy sobie ułatwiać życie w ten sposób, że gdy zaczynasz pisać jakąś procedurę, to od razu po jej nazwie wstawiasz oba nawiasy.

```
int procedura(void)
{
}
```

A dopiero w dalszej kolejności wypełniasz miejsce pomiędzy nawiasami. Dlaczego ? Ano dlatego, że jeżeli nabędziesz takiego zwyczaju, to unikniesz wielu godzin ślęczenia nad programem który nie daje się skompilować z powodu braku „głupiego" zamknięcia nawiasu. To samo dotyczy wszystkich innych nawiasów – zwykłych i kwadratowych.

W naszym przykładzie , przed nawiasem klamrowym zamykającym ciało funkcji *main()* znajduje się : *return (0)*; Jest to jedyna INSTRUKCJA zawarta w funkcji main(). Oznacza ona „zwróć" 0. W efekcie jej wykonania , funkcja main zwróci do systemu operacyjnego 0 i działanie programu zostanie zakończone.

Tutaj konieczne jest pewne wyjaśnienie – kompilator *AVR-GCC* wywodzi się Unixowego *GCC* i właśnie dlatego funkcja *main()* musi „zwracać" liczbę typu int. W przypadku mikrokontrolerów , nie ma żadnego systemu operacyjnego, do którego wykonywana aplikacja ma zwrócić efekt swojego działania. W efekcie program musi być tak napisany , że linia nr 4 nie może zostać nigdy wykonana, bo nie ma dokąd wracać z funkcji *main()* !!! Trochę dziwne prawda ? Ale tak po prostu jest i należy o tym wiedzieć.

Jak już wiesz, że z funkcji *main()* program nie ma dokąd wrócić, to możesz temu zaradzić w prosty sposób – wstawić do funkcji *main()* nieskończoną pętlę programową. Nieskończona pętla wygląda następująco:

```
 for(;;)
{
// tutaj instrukcje jakie mają być wykonywane
// albo zostaw puste , jeżeli mikrokontroler ma nic nie robić
}
```

Wracamy teraz do naszego pierwszego programu:

```
#include <avr/io.h>
#define LED_ON sbi(DDRB,PB1);sbi(PORTB,PB1)

int main (void)
{
LED_ON;

   for (;;)
  {

  }

   return (0);
}
```

To, co znajduje się na samym początku *#include <avr/io.h>* jest informacją dla kompilatora, że ma skorzystać z gotowych opisów i definicji mikrokontrolerów rodziny *AVR* . Jest to niezbędne aby proces kompilacji mógł się w ogóle odbyć.

W drugiej linii dokonujemy *MAKRODEFINICJI*, polega ona na tym że za pomocą dyrektywy *#define* informujemy kompilator, że dalej pojawi się nazwa makrodefinicji a po niej, oddzielone spacją poszczególne instrukcje wchodzące w skład tworzonej właśnie makrodefinicji. W naszym przypadku kompilator odczyta *#define* a następnie *LED_ON* – będzie w tym momencie wiedział, że chcemy stworzyć makrodefinicję o nazwie *LED_ON*. Następnie kompilator wczyta wszystko do znaku ; (średnika) i potraktuje to jako pierwszą instrukcję wchodzącą w skład makrodefinicji (u nas jest to ***sbi(DDRB,PB1)***. W następnym kroku kompilator wczytuje kolejne instrukcje, aż do momentu zakończenia linii. W naszej definicji makroinstrukcji jest to ***sbi(PORTB,PB1)***. W tym momencie makrodefinicja jest gotowa do zastosowania podczas dalszego pisania programu. Wystarczy napisać *LED_ON;* , a w procesie kompilacji zostanie to zastąpione na ***sbi(DDRB,PB1)***;***sbi(PORTB,PB1)***;

Po co komu makrodefinicje – przecież można nasz program napisać tak:

```
#include <avr/io.h>
int main (void)
{
sbi(DDRB,PB1);
```

```
sbi(PORTB,PB1);

   for (;;)
  {
  }
   return (0);
}
```

I będzie o wiele prościej ! Prościej będzie ,ale tylko na chwilę, do czasu modyfikacji programu. Wyobraź sobie, że masz program wyświetlający grafikę na wyświetlaczu LCD i zbudowałeś nowe urządzenie, w którym wyświetlacz jest podłączony do mikroprocesora trochę inaczej jak w rozwiązaniu dla którego napisałeś wcześniej stosowne oprogramowanie. Czeka Cię teraz wyszukiwanie i wymienianie wszystkich instrukcji sterujących liniami transmisyjnymi do wyświetlacza w całym programie! A ile błędów przy tym popełnisz – przekonasz się sam. Dobra szkoła programowania mikrokontrolerów jednoznacznie mówi:

Wszystkie podłączenia hardwareowe definiuj zawsze za pomocą makrodefinicji.

Dla silniejszego przekonania Cię o słuszności tego postępowania, powiem , że przeciętny sterownik mikroprocesorowy posiada kilkadziesiąt wejść i wyjść i dokonywanie zmian w kilkuset miejscach kodu źródłowego zajmuje wiele cennego czasu, który mógłbyś na przykład poświęcić na czytanie tej książki.

Instrukcje dostępu do portów

Niezależnie od tego , czy zastosujemy makrodefinicje, czy też wpiszemy instrukcje bezpośrednio do ciała funkcji *main()* , zastosowane *INSTRUKCJE* będą wyglądały następująco:

```
sbi(DDRB,PB1);
sbi(PORTB,PB1);
```

Instrukcje , są to komendy języka C, które zostaną w procesie kompilacji przetłumaczone na wewnętrzny język stosowanego mikroprocesora. Podane wyżej instrukcje są specyficzne dla rodziny AVR i dlatego nie znajdziesz ich omówienia w typowych podręcznikach języka C.

Instrukcja ***sbi(DDRB,PB1) ustawienie pina jako wyjścia.***

Ustawia ono podany pin mikroprocesora jako wyjście. Po resecie mikrokontrolera , wszystkie porty i piny są ustawione jako wejścia (z pewnymi wyjątkami).

Składnia jest następująca:

sbi(DDRx,Pxy);
Gdzie:
DDRx ma postać:
DDRA dla portu A
DDRB dla portu B
DDRC dla portu C
DDRD dla portu D
DDRE dla portu E
DDRG dla portu G

y ma postać: 0,1,2,3,4,5,6,7 i odpowiada numerowi pina danego portu.

Instrukcja ***sbi(PORTB,PB1) ustawienie wyjścia na stan wysoki.***

Ustawia wybrany pin na stan wysoki. Pin musi być wcześniej ustawiony w tryb pracy jako wyjście instrukcją **sbi**(PORTB,PB1). Po wykonaniu tej instrukcji mikrokontroler podaje na wybrany pin napięcie zasilania, jednocześnie ograniczając wartość wypływającego prądu do około 20 mA.

Składnia jest następująca:

sbi(PORTx,Pxy);
Gdzie:
PORTx ma postać:
PORTA dla portu A
PORTB dla portu B
PORTC dla portu C
PORTD dla portu D
PORTE dla portu E
PORTF dla portu F
PORTG dla portu G

Konfigurowanie kompilatora.

Teraz , gdy wiemy już wszystko na temat napisanego właśnie pierwszego programu musimy dokonać jego kompilacji i załadować skompilowany plik wykonywalny do pamięci mikroprocesora.

Procesem kompilacji steruje plik o nazwie *„makefile”* – bez żadnego rozszerzenia, po prostu *„makefile”*. Plik ten musi znajdować się w katalogu z kodem źródłowym w naszym wypadku w *C:\AVR_projekty\LED*. Można go utworzyć , tworząc w środowisku IDE nowy plik i wpisując następującą zawartość:

Zawartość pliku „**makefile**”:

```
PRG            = led
OBJ            = led.o
MCU_TARGET     = atmega8
OPTIMIZE       = -O2

DEFS           =
LIBS           =
```

Wpisywanie całej powyższej zawartości do pliku *„makefile”* to dosyć syzyfowa praca. Jest prostsze rozwiązanie – przekopiowanie i modyfikacja istniejącego już pliku „makefile”. Przejdź do katalogu *C:\WinAVR\examples\demo*. Odnajdź plik „makefile” i przekopiuj go do : *C:\AVR_projekty\LED*. Następnie otwórz go w edytorze IDE . Na początku pliku *„makefile”* zobaczysz coś takiego:

```
PRG            = demo
OBJ            = demo.o
MCU_TARGET     = at90s2313
```

Dokonaj zmian , tak aby powyższe linie wyglądały następująco:

```
PRG         = led
OBJ         = led.o
MCU_TARGET    = atmega8
```

Mrugająca dioda LED

Uzupełnij plik program led.c , aby zawierał poniższą treść. Jeżeli coś wyda Ci się niejasne, to nie wnikaj w istotę problemu. Po prostu przepisz to co znajduje się poniżej. Na wyjaśnienia przyjdzie czas za chwilę.

```
#include <avr/io.h>
/*********************************************************
                    Definicje stałych
*********************************************************/
#define F_CPU       1000000                                 /* 1MHz zegar procesora */
#define CYCLES_PER_US ((F_CPU+500000)/1000000) /* cpu cycles per microsecond */
/*************************************************************
                                Koniec definicji stałych
*************************************************************/

//definiujemy stany portu sterującego diodą LED
#define LED_ON sbi(DDRB,PB1);sbi(PORTB,PB1)
#define LED_OFF sbi(DDRB,PB1);cbi(PORTB,PB1)

// Piszemy procedury opóźnienia czasowego
void delay(unsigned int us)
{
   unsigned int delay_loops;
   register unsigned int  i;
   delay_loops = (us+3)/5*CYCLES_PER_US; // +3 for rounding up (dirty)
   for (i=0; i < delay_loops; i++) {};
}
void delayms(unsigned int ms)
{
        unsigned int i;
        for (i=0;i<ms;i++)
                {
                delay(999);
                asm volatile (
                "WDR"::);
                                }
}
int main (void)
{
     for (;;)
             {
                 LED_ON;
                 delayms(1000);
                 LED_OFF;
                 delayms(1000);
             }

   return (0);
}
```

Po wpisaniu powyższej treści skompiluj plik i załaduj do procesora, dioda powinna mrugać z czasem świecenia i przerwy wynoszącym jedną sekundę. Dioda nich sobie mruga, a my zajmiemy się wyjaśnieniem nowych elementów jakie pojawiły się w naszym programie. Na początku widzimy :

```
/*****************************************************
                    Definicje stałych
*****************************************************/
```

Jest to komentarz, czyli informacja, którą programista pisze dla siebie samego lub dla kolegów. Komentarz zaczyna się znakami /* a kończy znakami */ . Wszystko co znajdzie się pomiędzy nimi jest widoczne tylko dla człowieka, bo kompilator zupełnie ignoruje zawartość komentarzy. Duża ilość zastosowanych gwiazdek ma na celu ułatwienie odszukania istotnego komentarza podczas szybkiego przewijania okna z kodem źródłowym. Komentarz wygląda następująco:

```
/*
To jest pierwsza linia komentarza
A to jest druga linia
*/
```

Innym sposobem umieszczania komentarzy jest zastosowanie dwóch ukośników // . Są one stosowane jeżeli komentarz ma być krótki, mieszczący się w jednej linii programu. Przykładem takiego komentarza może być:

```
sbi(DDRB,PB1);  // ustawiamy pin PB1 jako wyjście
```

Komentarze , oprócz zastosowania do tworzenia opisów oprogramowania mają jeszcze inne, nie mniej ważne znaczenie. Pozwalają programiście na wyłączenie części kodu z napisanego programu, bez jej usuwania. Z tej możliwości będziesz często korzystał podczas uruchamiania oprogramowania. W naszym programie pojawiają się teraz dobrze już nam znane makrodefinicje:

```
#define F_CPU       1000000
#define CYCLES_PER_US ((F_CPU+500000)/1000000)
```

Pierwsza z nich definiuje F_CPU jako liczbę równą 1000000, co odpowiada ilości instrukcji wykonywanych przez mikrokontroler w ciągu jednej sekundy, przy zegarze systemowym ustawionym na 1MHz. Wszystkie, nowo zakupione mikrokontrolery AVR, jeżeli posiadają wbudowany zegar RC, to jest ona fabrycznie ustawiony na 1 MHz. Druga makrodefinicja definiuje CYCLES_PER_US jako coś co kompilator powinien obliczyć podanym dalej wzorem. Wyliczenie wartości CYCLES_PER_US jest wykonywane przez kompilator w trakcie kompilacji i dalej do programu jest podstawiana odpowiednia liczba. Oczywiście, można napisać funkcję która pozwoli na mikrokontrolerowi na obliczanie wymaganej wartości, tylko po co ? Skoro może to zrobić jednorazowo kompilator i odciążyć mikrokontroler od zbędnych obliczeń. Pętla programowa – wykonywanie zadania zadaną ilość razy.

```
void delay(unsigned int us)
{
   unsigned int delay_loops;
   register unsigned int  i;
   delay_loops = (us+3)/5*CYCLES_PER_US;
   for (i=0; i < delay_loops; i++) {};
}
```

W powyższym przykładzie zostaje utworzona procedura o nazwie *delay()*, znajdujące się przed nazwą procedury słowo void informuje kompilator, że procedura nie zwraca żadnych danych . W

nawiasie zwykłym , występującym po nawie procedury, jest wpisane *unsigned int us* – jest to informacja, że do procedury, podczas jej wywoływania trzeba przekazać argument typu unsigned int**.** Argument ten będzie następnie widziany we wnętrzu nowo utworzonej funkcji pod nazwą *us* . Nawias { otwiera ciało funkcji. Następnie są deklarowane zmienne , które będą używane w tej procedurze. Pierwszą deklarowaną zmienną jest delay_loops jak wynika z poprzedzających ją słów, jest ona typu *unsigned int* . Drugą deklarowaną zmienną jest zmienna o nazwie **i** . Jej deklaracja rozpoczyna się od słowa register. Użycie słowa register, powoduje , że do zapamiętywania tej zmiennej będą przeznaczone rejestry mikroprocesora. Normalnie, to znaczy bez użycia słowa register , dla zmiennej jest przydzielane miejsce w pamięci *RAM*. Podobnie jak pierwsza zmienna, zmienna i również jest typu unsigned int .

Dlaczego niektóre zmienne warto deklarować jako register ? Deklaracja zmiennej jako **register**, czyli tak , że zostaje dla niej przeznaczone miejsce w rejestrach mikroprocesora, mniej więcej dwukrotnie przyśpiesza szybkość operacji wykonywanych na tej zmiennej. Tak więc jeżeli zależy Ci na szybkim wykonaniu dużej ilości operacji na jakiejś zmiennej, a ma to miejsce w przypadku stosowania pętli programowych, to warto rozważyć zadeklarowanie zmiennej jako register. Istnieje tutaj pewne ograniczenie – rejestrów mikroprocesor nie ma zbyt wiele , bo tylko 32 i próba zadeklarowania kilku zmiennych jako register raczej się nie uda. Wracamy teraz do naszego programu. Kolejna linia:

```
delay_loops = (us+3)/5*CYCLES_PER_US;
```

dokonuje wyliczenia ile razy musi zostać wykonana pusta pętla programowa , aby uzyskać opóźnienie równe 1 mikrosekundzie. Podczas kompilacji tej linii , kompilator wstawi zamiast *CYCLES_PER_US* wartość liczbową , którą uzyska z poczynionej na wstępie makrodefinicji. Wartość *us* będzie przekazana do procedury w momencie jej wywoływania. W praktyce takie wywołanie wygląda następująco: *delay(400)*; - procedura *delay()* zostaje wywołana z parametrem 400 – oznaczającym wymagane opóźnienie w mikrosekundach. Przychodzi teraz kolej na wykonanie pustej pętli wyliczoną ilość razy. Przypominam, że wyliczona ilość jest już w zmiennej *delay_loops***.**

```
for (i=0; i < delay_loops; i++) {};
```

Powyższa linia oznacza to samo co:

```
for (i=0; i < delay_loops; i++)
{
}
```

Zauważ , że jeżeli cała procedura pętli została zapisana w jednej linii, to po zamykającym nawiasie klamrowym pojawił się średnik, którego nie ma jeżeli procedura jest zapisywana w czterech liniach. Poznałeś już wcześniej zapis pętli wykonywanej nieskończoną ilość razy. Od razu widać podobieństwo pętli wykonywanej zadaną ilość razy do tej poznanej wcześniej . W obu przypadkach konstrukcja jest ta sama:

```
for(warunek początkowy ; warunek trwania ; powiększenie licznika pętli)
{
}
```

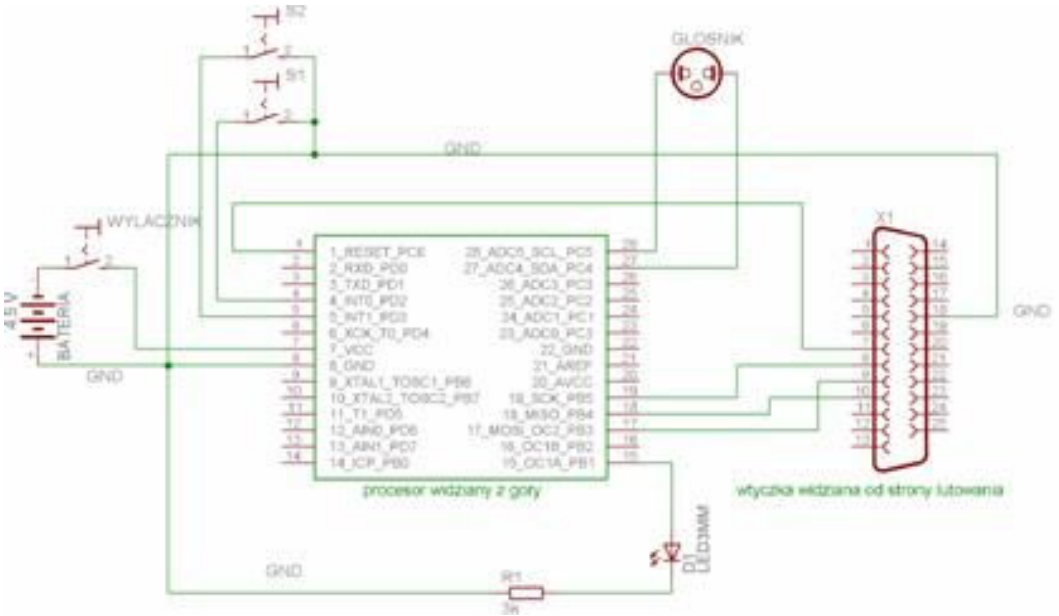
W naszym przypadku instrukcję pętli programowej można przeczytać w następujący sposób: „dla wszystkich i < od zawartości zmiennej delay_loops, począwszy od i=0. Występujący na końcu zapis i++ oznacza powiększenie wartości zmiennej i po każdym wykonaniu pętli. Jeżeli byś chciał aby wartość i ulegała powiększeniu przed wykonaniem zawartości pętli to napisz *++i* zamiast *i++*. Następnie w programie jest definiowana kolejna procedura *delayms()*. Jest ona bardzo podobna do objaśnianej wcześniej procedury *delay()*. Wyjaśnienia wymaga jedynie linia :

```
asm volatile (  "WDR"::);
```

Jest to wstawienie kodu asemblera w program w języku C. Wstawiana jest instrukcja asemblera WDR.

Grajek, czyli obsługa klawiatury

Jak dotąd, wszystkie dotychczasowe programy wykorzystywały możliwości mikrokontrolera *ATMega*8 jedynie w zakresie sterowania wyjściami. Teraz napiszemy program w którym mikrokontroler będzie reagował na naciskanie przycisków klawiatury. Zrobimy tak, aby naciśniecie przycisku powodowało wydanie przez głośnik określonego dźwięku. Najpierw trzeba nieco rozbudować nasz układ, podłączamy przyciski *S1* i *S2* odpowiednio do *PD2* i *PD3*.



Czynności przy tworzeniu programu są analogiczne do wcześniejszych. Tworzymy podkatalog *Grajek* i kopiujemy do niego pliki z katalogu Beep. Nazwę pliku **beep.c** zmieniamy na **grajek.c** zmieniamy również **beep.h** na **grajek.h** i dokonujemy niezbędnych zmian w pliku makefile oraz **zaprogramuj.bat**. Nie zapominamy również o zmianie parametrów narzędzia Zaprogramuj w środowisku IDE. Następnie procedurę beep() wycinamy z pliku **beep.c** i umieszczamy ją w pliku **poprzednie.c**. W efekcie tych działań , nasz plik **grajek.c** wygląda następująco:

```
#include <avr/io.h>
#include "beep.h"
#include "poprzednie.c"

int main (void)
{
    for (;;)
   {
                    LED_ON;
                    beep(20,100);
                    beep(2,1000);
                    delayms(1000);
                    LED_OFF;
                    beep(30,70);
                    delayms(1000);
```

```
  }
   return (0);
}
```

Dokonujemy zmian w pliku **grajek.c**, tak aby uzyskać następującą zawartość pliku:

```
#include <avr/io.h>
#include "grajek.h"
#include "poprzednie.c"

unsigned char klawiatura(void)
{
unsigned char zwrot=0;
// najpierw ustawiamy piny jako wyjścia na wysokim poziomie logicznym
// robimy to po to aby przeładować pojemności wejściowe mikrokontrolera
// bo nie stosujemy rezystorów podciągających wejścia a wejścia mają dużą
// rezystancję wejściową i zbierają wszystkie zakłócenia !

sbi(klawisz_s1_DDR,klawisz_s1_pin);
sbi(klawisz_s2_DDR,klawisz_s2_pin);
sbi(klawisz_s1_port,klawisz_s1_pin);
sbi(klawisz_s2_port,klawisz_s2_pin);

//dajemy 5 mikrosekund na przeładowanie
delay(5);
// teraz przełączamy piny w tryb pracy jako wejścia
cbi(klawisz_s1_DDR,klawisz_s1_pin);
cbi(klawisz_s2_DDR,klawisz_s2_pin);
// sprawdzamy stan wejść
if(bit_is_clear(klawisz_s1_wejscie,klawisz_s1_pin))
{
zwrot=zwrot+1;
}
if(bit_is_clear(klawisz_s2_wejscie,klawisz_s2_pin))
{
zwrot=zwrot+2;
}
// tutaj możesz dopisać obsługę kolejnych klawiszy
// muszą one dodawać do zmiennej zwrot kolejno 4 8 16 32 64 128
// takie wagi klawiszy powlają na rozróżnianie wciśniętych
//kombinacji wielo klawiszowych
return zwrot;
}

void zagraj_stan_klawiszy(void)
{
if(klawiatura()==1)
                {
                beep(30,70);
                }
if(klawiatura()==2)
                {
                beep(40,70);
                }
if(klawiatura()==3)
                {
```

```
                    beep(50,70);
                    }
}


int main (void)
{
   for (;;)
  {
            LED_ON;
                    delayms(100);
                    zagraj_stan_klawiszy();
                    LED_OFF;
                    delayms(100);
                    zagraj_stan_klawiszy();
  }
   return (0);
}
```

Do pliku grajek.h dodajemy prototypy procedur:

```
unsigned char klawiatura(void);
void zagraj_stan_klawiszy(void);
```

Kompilujemy program i ładujemy do procesora. Naciśnięcie dowolnego przycisku generuje odpowiadający mu dźwięk. Wciśnięcie obu klawiszy jednocześnie generuje trzeci dźwięk. Przeanalizuj utworzoną w tym programie procedurę obsługi klawiatury. Zastosowałem w niej pewien dodatkowy zabieg programistyczny, polegający na ustawianiu pinów mikrokontrolera jako wyjść o wysokim poziomie logicznym. Dzięki temu nie musimy w naszym urządzeniu stosować rezystorów podciągających wejścia w celu uzyskania prawidłowego odczytu klawiatury. Jeżeli usuniesz te dwie linie odpowiadające za przeładowanie wejść, to na biurku układ będzie dalej funkcjonował prawidłowo. Problemy zaczną się kiedy zastosujesz taką procedurę obsługi klawiatury w prawdziwym sterowniku. Od czasu do czasu będzie wyprawiał on dzikie harce spowodowane zakłóceniami na wejściach klawiatury. Poeksperymentuj trochę z przewodami o długości kilkudziesięciu centymetrów, podłączonymi do klawiszy i telefonem komórkowym w pobliżu. Zapewne dojdziesz do ciekawych wniosków. Pamiętaj o tym aby zawsze stosować przeładowanie wejść przed ich odczytem albo stosować rezystory podciągające!

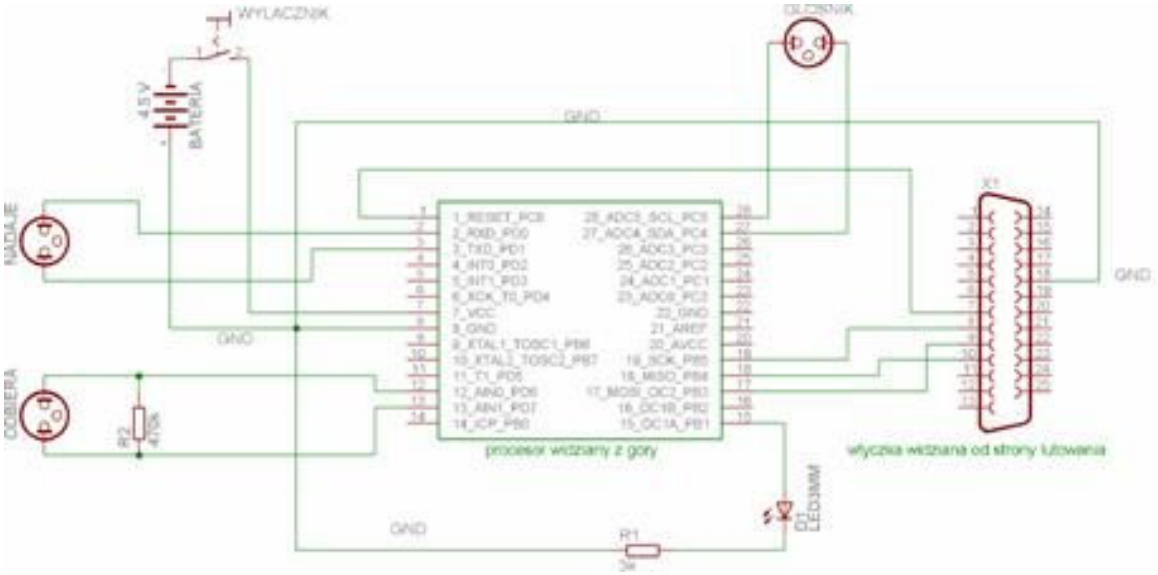
Sonar – wykorzystanie komparatora analogowego

W tym projekcie zbudujemy prosty sonar wykrywający przeszkody znajdujące się w otoczeniu. Sonar zwany równiach echolokatorem, wykorzystuje do swojego działanie fakt, że fale ultradźwiękowe doskonale odbijają się od większości przedmiotów. Sonar, podobnie jak nietoperz najpierw wysyła wiązkę ultradźwięków a następnie nasłuchuje powracające echo. Na podstawie opóźnienia echa jest obliczana odległość od przeszkody. Na podstawie siły powracającego sygnału można wysnuć wnioski dotyczące powierzchni, która odbiła ultradźwięki. Im twardsza, tym echo silniejsze. Sonary są powszechnie stosowane na łodziach podwodnych i w rybołówstwie, można je również spotkać w medycynie, gdzie pełnią rolę ultrasonografów. Współczesne sonary do przetwarzania odbitego echa wykorzystują specjalizowane procesory *DSP*, my jednak spróbujemy zbudować działający model sonaru na bazie prostego mikrokontrolera *ATMega8*! Nasze urządzenie ma następujące zadania do spełnienia: podczas normalnej pracy wysyłać regularnie wiązkę ultradźwięków i sygnalizować błyśnięciem diody stan czuwania. W razie odebrania wiązki od przeszkody znajdującej się bliżej niż zostało to określone w programie głośnik ma wydać dźwięk. Częstotliwość powtarzania dźwięków

ostrzegawczych ma wzrastać w miarę przybliżania się obiektu do sonaru. Gdy obiekt znajduje się bliżej niż 10cm głośnik wydaje dźwięk przypominający syrenę alarmową. Nasz sonar może być przydatny do parkowania samochodu lub budowy robota. Wystarczy umieścić go w zderzaku a głośniczek umieścić w kabinie.

Zasada działania komparatora analogowego.

Wbudowany komparator posiada dwa wejścia: *AIN0* i *AIN1* . Jeżeli napięcie na wejściu *AIN0* jest wyższe jak na wejściu *AIN*1 , to wyjście komparatora przyjmuje stan wysoki. O tym jaki jest aktualnie stan wyjścia możemy się dowiedzieć odczytując stan bitu *ACO* w rejestrze *ACSR*. Wykorzystamy tą właściwość do odbioru fal ultradźwiękowych. Przetwornik będzie podłączony bezpośrednio do wejścia komparatora. Ze względu na dużą oporność wyjściową przetwornika podłączamy do niego rezystor 470k w celu eliminacji zakłóceń. Gdy do przetwornika dotrą odbite fale ultradźwiękowe, to spowodują one pojawienie się na wyjściu przetwornika przebiegu sinusoidalnego. Sygnał sinusoidalny podany na wejście komparatora spowoduje zmianę stanu jego wyjścia co pół okresu sinusoidy. Wynika to z faktu, że przez pół okresu sinusoidy napięcie na wejściu *AIN0* jest wyższe od napięcia na wejściu *AIN1*, zaś w kolejnym półokresie sinusoidy polaryzacja napięcia się odwraca i napięcie na wejściu *AIN0* jest z kolei niższe od napięcia na wejściu *AIN1*. Najpierw jak zawsze modyfikujemy nieco nasz projekt. Tym razem dodajemy do niego dwa przetworniki ultradźwiękowe jak na poniższym schemacie:



W projekcie zastosowałem dwa przetworniki znalezione gdzieś w szufladzie i tak naprawdę to oba były z literką T , czyli nadawcze. Prawdopodobnie, jeżeli do odbioru ultradźwięków zastosuje się przetwornik z literką R, czyli odbiorczy to zasięg sonaru wzrośnie. Robiłem też próby z pracą na jednym przetworniku ale nie przyniosły one spodziewanego rezultatu. Po zaprzestaniu generowania fali ultradźwiękowej przetwornik potrzebuje dosyć dużo czasu na zaprzestanie drgań rezonansowych, nie pomagało nawet dodatkowe obciążanie przetwornika poprzez jednoczesne zwieranie obu jego nóżek do masy. W tej sytuacji odbiór fal odbitych mógł się rozpoczynać z dużym opróżnieniem, czyli dla dużych odległości. Z kolei fale odbite z dużej odległości mają tak znikomą energię , że nie są w stanie wyzwolić komparatora.

Układ z jednym przetwornikiem wymaga wzmocnienia sygnału i może mierzyć jedynie duże odległości, przy bliskich pomiarach staje się bezużyteczny. Po uruchomieniu układu może się zdarzyć, że sygnalizuje on fałszywe echo, możesz temu zaradzić odchylając nieco wyloty przetworników od wspólnej osi.

Po zmajstrowaniu powyższego układu, zabieramy się do programowania. Tworzymy nowy podkatalog projektu o nazwie Sonar i kopiujemy do niego co trzeba z projektu Grajek. Zawartość plików dostosowujemy do aktualnego projektu. Odpowiednio zmieniamy nazwy i zawartość plików oraz procedurę Zaprogramuj w *IDE*. Wszystko co zbędne w pliku sonar.c przenosimy do pliku poprzednie.c. Czyścimy zawartość procedury *main()*. Jak to zrobić jest opisane w projekcie Grajek. Po dokonaniu tych czynności nasz plik sonar.c powinien wyglądać następująco:

```
#include <avr/io.h>
#include "sonar.h"
#include "poprzednie.c"

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

Teraz wpisujemy do pliku *sonar.c* następującą treść:

```
#include <avr/io.h>
#include "sonar.h"
#include "poprzednie.c"


int ping(void)
{
// procedura wysyłająca paczkę ultradźwięków
int echo=0; // tutaj zanotujemy opóźnienie powrotu echa
sinus_40_khz();
sinus_40_khz();
sinus_40_khz();
sinus_40_khz();
sinus_40_khz();
sinus_40_khz();
sinus_40_khz();
sinus_40_khz();
sinus_40_khz();
sinus_40_khz();
// po wysłaniu paczki ultradźwięków wytłumiamy drgania
// przetwornika zwierając je do masy
cbi(ultrasonic_pinA_port,ultrasonic_pinA_pin);
cbi(ultrasonic_pinB_port,ultrasonic_pinB_pin);
delay(8);

//rozładowujemy pojemność przetwornika wejściowego
//poprzez zwarcie do masy
sbi(DDRD,PD6);
cbi(PORTD,PD6);
delay(8);
// włączamy wejście
cbi(DDRD,PD6);
// teraz czekamy na powrót echa

for(echo=0;echo<201;echo++)
{
if(bit_is_set(ACSR,ACO))
```

```
        {
                break;
        }
}
return echo;
}


void sinus_40_khz(void)
{
sbi(ultrasonic_pinA_DDR,ultrasonic_pinA_pin);sbi(ultrasonic_pinA_port,ultrasonic_pinA_pin);
sbi(ultrasonic_pinB_DDR,ultrasonic_pinB_pin);cbi(ultrasonic_pinB_port,ultrasonic_pinB_pin);
// teraz czekamy połowę okresu sinosoidy
asm volatile (
                "WDR"::);
asm volatile (
                "WDR"::);
asm volatile (
                "WDR"::);
asm volatile (
                "WDR"::);

cbi(ultrasonic_pinA_port,ultrasonic_pinA_pin);
sbi(ultrasonic_pinB_port,ultrasonic_pinB_pin);

// teraz czekamy nieco krócej jak okres połowy sinusoidy
// bo procesor potrzebuje nieco czasu na powrót z tej procedury
// i ponowne do niej wejście - dopiero wtedy rozpocznie kolejną
// sinusoidę

asm volatile (
                "WDR"::);
asm volatile (
                "WDR"::);

}

int main (void)
{
int odleglosc=0;
int dzwiek=0;
LED_ON;
 beep(2,150);
 beep(3,70);
 LED_OFF;

// zwieramy nieodwracające wejście komparatora do masy
sbi(DDRD,PD7);
cbi(PORTD,PD7);

for (;;)
  {
  LED_ON;
  delayms(10);
  LED_OFF;
  delayms(100);
        odleglosc=ping();
        if(odleglosc<10)
```

```
            {
            beep(2,50);
            beep(3,50);
            }
            if(odleglosc<200)
            {
            dzwiek=odleglosc;
            beep(1,dzwiek);
            }

  }
   return (0);
}
```

Do pliku sonar.h, który utworzyliśmy przez przekopiwanie pliku grajek.h i zmianę jego nawy, dodajemy prototypy nowoutworzonych funkcji:

```
int ping(void);
void sinus_40_khz(void);
```

Następnie wpisujemy makrodefinicje dotyczące podłączenia przetwornika nadawczego:

```
//definiujemy wyjścia do których jest podłączony nadajnik ultradźwięków
#define ultrasonic_pinA_port PORTD
#define ultrasonic_pinA_pin PD0
#define ultrasonic_pinA_DDR DDRD
// teraz drugi pin
#define ultrasonic_pinB_port PORTD
#define ultrasonic_pinB_pin PD1
#define ultrasonic_pinB_DDR DDRD
```

Kompilujemy program i ładujemy go do procesora. Sonar jest gotowy do użycia. Jak już się nim pobawisz, to przystąpimy do analizy kodu użytego przy budowie sonaru.

Wcześniej pisałem, że nieskończona pętla programowa, zapisywana jako:

```
for(;;)
{

}
```

Będzie wykonywana do czasu wyłączenia mikrokontrolera. Owszem, będzie o ile programista jej wcześniej nie przerwie! Można zawsze wyjść z tej jak i innych pętli poprzez użycie instrukcji **break**. Instrukcję **break** stosuje się najczęściej po wykryciu zaistnienia sytuacji , która była oczekiwana. Spójrz jak to zrobiłem w przykładowym sonarze:

```
.....
for(echo=0;echo<201;echo++)
{
if(bit_is_set(ACSR,ACO))
        {
                break;
        }
```

```
}
return echo;
}
```

Założyłem, że w pewnym przedziale czasowym powinno wrócić odbite echo, które spowoduje ustawienie wyjścia komparatora analogowego w stan wysoki. Dlatego też procedurę sprawdzania , czy wyjście komparatora jest w stanie wysokim umieściłem w pętli, wykonywanej maksymalnie 201 razy. Możesz to zmienić na więcej, ale później wracające echo jest już tak słabe, że komparator i tak nie zadziała. Jeżeli echo się nie pojawi, to procedura zwróci liczbę 201, bo tyle będzie wynosiła wartość licznika pętli po jej zakończeniu. Jeżeli jednak w czasie sprawdzania stanu wejścia komparatora zostanie wykryte, że ma ono stan wysoki, to zostanie wykonana instrukcja break, kończąca pracę pętli natychmiast. W tym momencie, licznik pętli, którym jest zmienna echo, zawiera ilość przejść już wykonanych przez pętlę *for()*. Wartość zmiennej echo jest zwracana jako miara opróżnienia do procedury, która ją wywołała. Często popełnianym błędem jest optymistyczna wiara programisty w to ,że echo zawsze powróci i wyzwoli komparator. Będąc takim optymistą , możesz powyższą pętle napisać w następujący sposób:

```
.....
echo=0;
for(;;)
{
if(bit_is_set(ACSR,ACO))
                                {
                                 break;
                                }
}
echo=echo+1; //powiększamy licznik opóźnienia
return echo;
}
```

Jeżeli jednak optymizm się nie sprawdzi i echo nie powróci, to mikroprocesor pozostanie w tej pętli naprawdę na zawsze! Pamiętaj, aby pisząc oprogramowanie zawsze rozważać nie tylko sytuację gdy pożądane zdarzenie zaistnieje, ale również o wiele ważniejszą sytuację, gdy zdarzenie na które oczekujesz nie zajdzie!

Listingi programów

W opracowaniu umieszczono listingi 11 programów, bez opisu, mających na celu przedstawienie podejścia do różnych problemów takich jak np. obsługa wyświetlacza LCD, komunikacji I2C i wiele innych.

```
/*******************************************************/
/* Ćwiczenie 1 - sterowanie portami w trybie wyjściowym  */
/* Efekty świetlne na linijce LED-ów                   */
/* J.D. '2003                                          */
/*******************************************************/

#include <io.h>
unsigned long pczekaj=1500;

void czekaj(unsigned long pt) //procedura wytracania czasu
{
 unsigned char tp1;

 for(;pt>0;pt--)
 {
  for(tp1=255;tp1!=0;tp1--);
 }}

int main(void) //program główny
{
 unsigned char ledy,i,licznik;

 DDRB=0xff;  //konfiguracja wszystkich wyprowadzeń
          //portu B jako wyjścia
 while(1)    //nieskończona pętla główna programu
 {
//efekt węża
  for(licznik=0;licznik<10;licznik++)//pętla długości
     //trwania efektu (liczba cykli danego efektu)
  {
   PORTB=0xff;   //wygaś LED-y
   czekaj(pczekaj);
   for(i=0;i<8;i++)         //pętla zmieniająca fazę efektu
   {
    cbi(PORTB,i);//wysteruj (zapal) pojedynczego LED-a
    czekaj(pczekaj);
   }
   for(i=0;i<8;i++)         //pętla zmieniająca fazę efektu
   {
   sbi(PORTB,i);  //wysteruj (zgaś) pojedynczego LED-a
    czekaj(pczekaj);
   } }

//efekt biegnącego punktu
  PORTB=0xff;    // wygaś LED-y
  for(licznik=0;licznik<10;licznik++)  //pętla długości
   //trwania efektu (liczba cykli danego efektu)
   {
    for(ledy=0xfe;ledy!=0xff;ledy=(ledy<<=1)+1)
      //pętla zmieniająca fazę efektu
```

```
  {
   PORTB=ledy; //wysterowanie LED-ów zgodne z wartością
  //zmiennej ledy
   czekaj(pczekaj);          //wytrać czas
  } } }}
```

```
/*****************************************************/
/* Ćwiczenie 2 - Obsługa timera0 w trybie odpytywania */
/* Generator fali prostokątnej 1kHz                  */
/*              wy: PORTB-8                          */
/* J.D. '2003                                        */
/*****************************************************/
#include <io.h>
#define tau0 6;//stała czasowa dla 1kHZ @8MHz

int main( void )
{
 unsigned char licznik=2;

 DDRB=0x01;     // wyjściem generatora będzie PB0
 TCNT0=tau0;    //wpisz stałą czasową dla zadanego interwału
 TCCR0=2;       //timer0 będzie pracował z preskalerem Fosc/8

 while(1)
 {
  while((inp(TIFR)&0x02)!=0x02);         //czekaj na ustawienie flagi
            //TOV0 (przekręcenie licznika)
  TCNT0=tau0;   //wpisz stałą czasową
  if(--licznik==0) //zmiana polaryzacji wyjścia wymaga
            //2-krotnego przekręcenia się licznika
  {
   PORTB^=0x01;          //zmień stan wyjścia
licznik=2;//odśwież stan licznika
  }
  TIFR=1<<TOV0; //kasuj flagę przepełnienia
 }
}
```

```
/*************************************************************/
/* Ćwiczenie 3 - sterowanie portami w trybie wejściowym      */
/* Obsługa przycisków dołączonych do portów mikrokontrolera */
/* Licznik rewersyjny naciśnięć klawiszy                     */
/* J.D.  '2003                                               */
/*************************************************************/
#include <io.h>
#include <interrupt.h>
#include <signal.h>

#define tau0 247;       //stała czasowa timera0
#define vliczt0 113;    //stała wpisywana do licznika wejść do
```

```
  //do przerwania timera0

//******************** zmienne globalne ******************
unsigned char liczt0;     //Licznik wejść do przerwania timera0.
  //Klawisz jest badany, gdy liczt0=0
unsigned char licznik;   //Licznik rewersyjny, którego stan jest
  //wyświetlany na linijce LED-ów

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

SIGNAL (SIG_OVERFLOW0)       //obsługa przerwania
  //od przepełnienia timera0
{
 TCNT0=tau0;          //odśwież stałą czasowa w TCNT0
 PORTD^=1<<PD6;      //zmień stan wy generatora 434Hz
 if(--liczt0==0)    //czy już czytać klawisze?
 {          //tak
  if(bit_is_clear(PIND,PD1))    //czytaj SW4 - <->
  {
   licznik--;
  }
  else
  {
   if(bit_is_clear(PIND,PD0))    //czytaj SW1 - <+>
   {
    licznik++;
   }
  }
  PORTB=~licznik;         //wyświetl stan licznika na LED-ach
  //negacja jest potrzebna, gdyż LEDY są
  //zapalane niskim stanem
 liczt0=vliczt0;    //odśwież stan liczt0
 }
}

int main(void)
{
 liczt0=vliczt0;

 DDRD=0x60;     //PD0-PD4 jako wejściowy, PD5-PD6 - wy
 PORTD=0xff;    //z podciąganiem
 DDRB=0xff;     //PORTB - wy
 PORTB=0xff;    //z podciąganiem
 TIMSK=1<<TOIE0;     //zezwolenie na przerwania od TC0
 TCNT0=tau0;          //wpisz stałą czasowa do TCNT0
 TCCR0=5;             //preskaler XTAL/1024,
  //kwant mierzonego czasu = 128<m>s
 sei();               //odblokuj globalne przerwania

 while(1)
 {
```

```
  PORTD^=0x20;          //generator
  czekaj(10000L);
 }
}
```

```
/*************************************************************/
/* Ćwiczenie 4a - sterowanie alfanumerycznym wyświetlaczem LCD */
/*              16x2 (16 znaków, 2 wiersze)                  */
/*            - obsługa pojedynczego klawisza                */
/* J.D. '2003                                                */
/*************************************************************/
#include <io.h>
#include <progmem.h>
#include <stdlib.h>

#define lcd_rs 2          //definicja bitu portu dla linii RS
#define lcd_e 3           //definicja bitu portu dla linii E
#define CR 0x0a           //definicja znaku CR (przejście do nowej linii)

unsigned char wiersz=0;
unsigned char kolumna=0;

void czekaj(unsigned long pt) //procedura wytracania czasu
{
#define tau 10.38         //przybliżony przelicznik argumentu na ms
 unsigned char tp1;

 for(;pt>0;pt--)
 {
  for(tp1=255;tp1!=0;tp1--);
 }
}

void piszilcd(unsigned char instr) //zapisz instrukcję sterującą do LCD
{
 cbi(PORTB,lcd_rs);
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|(instr&0xf0);  //przygotuj starszy półbajt do LCD
 asm("nop");                  //wymagane wydłużenie impulsu
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);             //impuls strobujący
 czekaj(10L);                  //czekaj na gotowość LCD ok. 100us
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|((instr&0x0f)<<4);  //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);             //impuls strobujący
 czekaj(10L);                  //czekaj na gotowość LCD ok. 100us
}

void piszdlcd(char dana)       //zapisz daną do LCD
{
 sbi(PORTB,lcd_rs);
```

```
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|(dana&0xf0);   //przygotuj starszy półbajt do LCD
 asm("nop");                       //wymagane wydłużenie impulsu
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);                 //impuls strobujący
 czekaj(10L);                      //czekaj na gotowość LCD
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|((dana&0x0f)<<4); //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);                 //impuls strobujący
 czekaj(10L);                      //czekaj na gotowość LCD
}

void czysclcd(void)                //czyść ekran
{
 piszilcd(0x01);      //polecenie czyszczenia ekranu dla kontrolera LCD
 czekaj(1.64*tau);    //rozkaz 0x01 wykonuje się 1.64ms
 wiersz=0;
 kolumna=0;
}

void piszznak(char znak)    //procedura umieszcza znak na wyświetlaczu
{
 piszdlcd(znak);            //wyświetl znak na LCD
 if(++kolumna==16)          //czy bieżąca kolumna mieści się na wyświetlaczu?
 {
  kolumna=0;                //jeśli nie, to ustaw początkową...
  if(++wiersz==2)           //i przejdź do nowego wiersza
  {
   wiersz=0;       //jeśli nowy wiersz jest poza wyświetlaczem, ustaw początkowy
  }
 }
}

void lcdxy(unsigned char w, unsigned char k)    //ustaw współrzędne kursora
{
 piszilcd((w*0x40+k)|0x80);  //standardowy rozkaz sterownika LCD
                             //ustawiający kursor w określonych współrzędnych
}

void pisztekst(char *tekst)  //pisz tekst na LCD wskazywany pointerem *tekst
{
 char zn;
 char nr=0;

 while(1)
 {
  zn=PRG_RDB(&tekst[nr++]);  //pobranie znaku z pamięci programu
  if(zn!=0)                 //czy nie ma końca tekstu?
  {
   if(zn==CR)               //czy znak nowej linii
   {
    wiersz==1?wiersz=0:++wiersz; //przejdź do nowej linii
    kolumna=0;
    lcdxy(wiersz,kolumna);   //ustaw obowiązujące po zmianie współrzędne na LCD
```

```
  }
  else
  {
   piszdlcd(zn);          //umieść pojedynczy znak tekstu na LCD
  }
 }
 else
 {
  break;                 //zakończ pętlę, jeśli koniec tekstu
 }
}
}

void klawisz(unsigned char pozkl)  //czekaj na naciśnięcie określonego klawisza
                           //klawisz jest wybierany poprzez argument pozkl
{
 while(PIND&pozkl);              //czekaj na naciśnięcie klawisza
 czekaj(30*tau);                //odczekaj aż ustaną zakłócenia
 while(!(PIND&pozkl));           //czekaj na zwolnienie klawisza
}

int main(void) //program główny
{
 unsigned char i;
 //>>>>>>>>>>>>>    definicje tekstów wyświetlanych na LCD   <<<<<<<<<<<<<
 char *tekst1=PSTR("kursor, mruga");
 char *tekst2=PSTR("bez kursora\nbez mrugania");
                           // \n oznacza znak CR (przejście do nowej linii)
 char *tekst3=PSTR("kursor\nbez mrugania");
 char *tekst4=PSTR("bez kursora\nmruganie");
 char *tekst5=PSTR("Przesuwanie\ntekstu w prawo");
 char *tekst6=PSTR("Przesuwanie\ntekstu w lewo");
 char *tekst7=PSTR("pozycja 1,3");
 char *tekst8=PSTR("pozycja 0,9");
 char *tekst9=PSTR("Hello World!");

 //>>>>>>>>>>>>>>>>>>>    konfigurowanie portów
<<<<<<<<<<<<<<<<<<<<<<<<<<
 PORTD=0xff;   //port z podciąganiem
 PORTB=0x03;   //port z podciąganiem
 DDRD=0x00; //PORTD - we
 DDRB=0xff; //PORTB - wy

 //>>>>>>>>>>>>>>>>>   inicjacja wyświetlacza LCD   <<<<<<<<<<<<<<<<<<<<<<
 czekaj(45*tau);        //opóźnienie ok. 45ms dla ustabilizowania się napięcia
                        //zasilania LCD (katalogowo min. 15 ms)
                        //lcd_rs już wcześniej było ustawione w stanie "0"
 for(i=0;i<3;i++)       //3-krotne wysłanie 3-
 {
  sbi(PORTB,lcd_e);
  PORTB=(PORTB&0x0f)|0x30; //wyślij 3- do LCD
  asm("nop");
  asm("nop");
  asm("nop");
  cbi(PORTB,lcd_e);
  czekaj(5*tau);      //ok. 5ms
 }
 sbi(PORTB,lcd_e);
```

```
 PORTB=(PORTB&0x0f)|0x20;  //wyślij 2- do LCD
 asm("nop");              //wymagane wydłużenie impulsu
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);
 czekaj(10L);        //od tego momentu można sprawdzać gotowość LCD
                  //w tym programie nie będzie sprawdzania gotowości

 piszilcd(0x28);     //interfejs 4-bitowy, 2 linie, znak 5x7
 piszilcd(0x08);     //wyłącz LCD, wyłącz kursor, wyłącz mruganie
 piszilcd(0x01);     //czyść LCD
 czekaj(1.64*tau);   //wymagane dla instrukcji czyszczenia ekranu opóźnienie
 piszilcd(0x06);     //bez przesuwania w prawo

 while(1)
 {
//>>>>>>>>>>>>  EFEKT 1  <<<<<<<<<<<<<<
  piszilcd(0x0f);    //włącz LCD, włącz kursor, włącz mruganie
  pisztekst(tekst1);
  klawisz(2);
  czysclcd();

//>>>>>>>>>>>>  EFEKT 2  <<<<<<<<<<<<<<
  piszilcd(0x0c);    //włącz LCD, bez kursora, bez mrugania
  pisztekst(tekst2);
  klawisz(2);
  czysclcd();

//>>>>>>>>>>>>  EFEKT 3  <<<<<<<<<<<<<<
  piszilcd(0x0e);    //włącz LCD, włącz kursor, bez mrugania
  pisztekst(tekst3);
  klawisz(2);
  czysclcd();

//>>>>>>>>>>>>  EFEKT 4  <<<<<<<<<<<<<<
  piszilcd(0x0d);    //włącz LCD, bez kursora, z mruganiem
  pisztekst(tekst4);
  klawisz(2);
  czysclcd();

//>>>>>>>>>>>>  EFEKT 5  <<<<<<<<<<<<<<
  pisztekst(tekst5); //umieść tekst na LCD, a następnie przesuń w prawo
  czekaj(1000L*tau);
  for(i=0;i<16;i++)
  {
   piszilcd(0x1c);   //przesuń
   czekaj(200*tau);
  }
  czysclcd();

//>>>>>>>>>>>>  EFEKT 6  <<<<<<<<<<<<<<
  pisztekst(tekst6);  //umieść tekst na LCD, a następnie przesuń w lewo
  czekaj(1000L*tau);
  for(i=0;i<16;i++)
  {
   piszilcd(0x18);   //przesuń
   czekaj(200*tau);
  }
```

```
  czysclcd();

//>>>>>>>>>>>>  EFEKT 7  <<<<<<<<<<<<<<
//umieszczanie kursora w różnych współrzędnych ekranu
  pisztekst(tekst7);
  lcdxy(1,3);
  klawisz(2);
  czysclcd();

  pisztekst(tekst8);
  lcdxy(0,9);
  klawisz(2);
  czekaj(500*tau);

//>>>>>>>>>>>>  EFEKT 8  <<<<<<<<<<<<<<
//wprowadzanie tekstu spoza ekranu
  piszilcd(0x0c);      //wyłącz kursor
  czysclcd();
  piszilcd(0x80|0x10); //ustaw współrzędne poza ekranem
  pisztekst(tekst9);   //pisz tekst poza ekranem...
  for(i=0;i<14;i++)    //i przesuń w lewo
  {
   piszilcd(0x18);     //przesuń
   czekaj(200*tau);
  }
  czekaj(200*tau);

//>>>>>>>>>>>>  EFEKT 9  <<<<<<<<<<<<<<
//mruganie tekstem wyświetlanym na LCD
//ze zmienną częstotliwością
  for(i=10;i!=0xff;i--)
  {
   piszilcd(0x08);   //wyłącz LCD
   czekaj(i*1038);
   piszilcd(0x0c);   //włącz LCD
   czekaj(i*1038);
  }
  czekaj(200*tau);
  klawisz(2);

  czysclcd();
  lcdxy(1,2);        //obsunięcie tekstu w dół
  pisztekst(tekst9);
  czekaj(500*tau);
  for(i=0;i<16;i++)  //zatarcie tekstu znak po znaku
  {
   lcdxy(1,i);
   piszznak(' ');
   czekaj(200*tau);
  }
  czysclcd();
 }
}
```

```
/*****************************************************************/
/* Ćwiczenie 4b - sterowanie alfanumerycznym wyświetlaczem LCD */
/*              16x1 (16 znaków, 1 wiersz)                       */
/*            - obsługa pojedynczego klawisza                    */
/* J.D. '2003                                                    */
/*****************************************************************/
#include <io.h>
#include <progmem.h>

#define lcd_rs 2
#define lcd_e 3
#define CR 0x0a

volatile unsigned char kolumna;

void czekaj(unsigned long pt) //procedura wytracania czasu
{
#define tau 10.38           //przybliżony przelicznik argumentu na ms
 unsigned char tp1;

 for(;pt>0;pt--)
 {
  for(tp1=255;tp1!=0;tp1--);
 }
}

void piszilcd(unsigned char instr) //zapisz instrukcję sterującą do LCD
{
 cbi(PORTB,lcd_rs);
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|(instr&0xf0);  //przygotuj starszy półbajt do LCD
 asm("nop");                       //wymagane wydłużenie impulsu
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);                 //impuls strobujący
 czekaj(10L);                      //czekaj na gotowość LCD ok. 100us
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|((instr&0x0f)<<4);  //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);                 //impuls strobujący
 czekaj(10L);                      //czekaj na gotowość LCD ok. 100us
}

void piszdlcd(char dana)           //zapisz daną do LCD
{
 sbi(PORTB,lcd_rs);
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|(dana&0xf0);   //przygotuj starszy półbajt do LCD
 asm("nop");                       //wymagane wydłużenie impulsu
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);                 //impuls strobujący
 czekaj(10L);                      //czekaj na gotowość LCD
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|((dana&0x0f)<<4); //przygotuj młodszy półbajt do LCD
 asm("nop");
```

```
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);              //impuls strobujący
 czekaj(10L);                   //czekaj na gotowość LCD
}

void czysclcd(void)             //czyść ekran
{
 piszilcd(0x01);      //polecenie czyszczenia ekranu dla kontrolera LCD
 czekaj(1.64*tau);    //rozkaz 0x01 wykonuje się 1.64ms
 kolumna=0;
}

void piszznak(char znak)    //procedura umieszcza znak na wyświetlaczu
{
 unsigned char p;
 if(kolumna<8)
 {
  p=kolumna;
 }
 else
 {
  p=kolumna+56;       //dodatkowe przesunięcie współrzędnych dla LCD 1x16
                      //dla drugiej połówki wyświetlacza
 }
 piszilcd(p|0x80);
 kolumna=kolumna!=15?++kolumna:0;  //ustaw następną kolumnę
 piszdlcd(znak);                   //wyświetl znak na LCD
}

void lcdy(unsigned char k)  //ustaw współrzędne kursora
{
 if(k<8)
 {
  piszilcd(k|0x80);         //standardowy rozkaz sterownika LCD
                            //ustawiający kursor w określonych współrzędnych
 }
 else
 {
   piszilcd((k+56)|0x80);   //dodatkowe przesunięcie współrzędnych dla LCD 1x16
 }
 kolumna=k;
}

void pisztekst(char *tekst)  //pisz tekst na LCD wskazywany pointerem *tekst
{
 char nr=0;
 char zn;

 while(1)
 {
  zn=PRG_RDB(&tekst[nr++]);  //pobranie znaku z pamięci programu
  if(zn!=0)                  //czy nie ma końca tekstu?
  {
   if(zn==CR)                //czy znak nowej linii
   {
    kolumna=0;       //jeśli tak, ustaw obowiązujące po zmianie współrzędne na LCD
   }
```

```
  else
  {
   piszznak(zn);          //umieść pojedynczy znak tekstu na LCD
  }
 }
 else
 {
  break;                 //zakończ pętlę, jeśli koniec tekstu
 }
}
}

void klawisz(unsigned char pozkl)  //czekaj na naciśnięcie określonego klawisza
                           //klawisz jest wybierany poprzez argument pozkl
{
 while((PIND&pozkl)==pozkl);       //czekaj na naciśnięcie klawisza
 czekaj(30*tau);                   //odczekaj aż ustaną zakłócenia
 while((PIND&pozkl)!=pozkl);       //czekaj na zwolnienie klawisza
}

int main(void) //program główny
{
 unsigned char i;

 //>>>>>>>>>>>>>    definicje tekstów wyświetlanych na LCD   <<<<<<<<<<<<<
 char *tekst1=PSTR("Wyswietlacz 1x16");
 char *tekst2=PSTR("Hello World!");

 //>>>>>>>>>>>>>>>>>>>    konfigurowanie portów
<<<<<<<<<<<<<<<<<<<<<<<<<<
 PORTD=0xff;   //port z podciąganiem
 PORTB=0x03;   //port z podciąganiem
 DDRD=0x00;    //PORTD - we
 DDRB=0xff;    //PORTB - wy

 //>>>>>>>>>>>>>>>>>>   inicjacja wyświetlacza LCD   <<<<<<<<<<<<<<<<<<<<<<
 czekaj(45*tau);        //opóźnienie ok. 45ms dla ustabilizowania się napięcia
                  //zasilania LCD (katalogowo min. 15 ms)
                  //lcd_rs już wcześniej było ustawione w stanie "0"
 for(i=0;i<3;i++)       //3-krotne wysłanie 3-
 {
  sbi(PORTB,lcd_e);
  PORTB=(PORTB&0x0f)|0x30; //wyślij 3- do LCD
  asm("nop");
  asm("nop");
  asm("nop");
  cbi(PORTB,lcd_e);       //impuls strobujący
  czekaj(5*tau);          //ok. 5ms
 }
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|0x20;  //wyślij 2- do LCD
 asm("nop");               //wymagane wydłużenie impulsu
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);         //impuls strobujący
 czekaj(10L);
 piszilcd(0x28);     //interfejs 4-bitowy, 2 linie, znak 5x7
 piszilcd(0x0c);     //wyłącz LCD, wyłącz kursor, wyłącz mruganie
```

```
 piszilcd(0x01);     //czyść LCD
 czekaj(1.64*tau);   //wymagane dla instrukcji czyszczenia ekranu opóźnienie
 piszilcd(0x06);     //bez przesuwania w prawo

 while(1)
 {
//>>>>>>>>>>>>   EFEKT 1   <<<<<<<<<<<<<<<
//wyświetl tekst
  pisztekst(tekst1);
  klawisz(2);
  pisztekst(tekst2);
  klawisz(2);
  czysclcd();

//>>>>>>>>>>>>   EFEKT 2   <<<<<<<<<<<<<<<
//umieszczanie kursora w różnych współrzędnych ekranu
  piszilcd(0x0e);  //włącz LCD, włącz kursor bez mrugania
  for(i=0;i<8;i++)
  {
   lcdy(i);          //umieść kursor
   czekaj(500*tau);
   czysclcd();
   lcdy(15-i);       //umieść kursor
   czekaj(500*tau);
   czysclcd();
  }

//>>>>>>>>>>>>   EFEKT 3   <<<<<<<<<<<<<<<
//mruganie tekstem wyświetlanym na LCD
  czysclcd();
  pisztekst(tekst2); //wyświetl tekst na LCD

  for(i=0;i<20;i++)
  {
   piszilcd(0x08);   //wyłącz LCD
   czekaj(300*tau);
   piszilcd(0x0c);   //włącz LCD
   czekaj(300*tau);
   if(!(PIND&0x02))
   {
    break;           //zakończenie pętli po naciśnięciu SW4
   }
  }
  czysclcd();
  czekaj(200*tau);
 }
}
```

```
/*****************************************************************/
/* Ćwiczenie 5 - Obsługa klawiatury matrycowej z wykorzystaniem */
/*              przerwań timera0                                 */
/*           - wykorzystanie wyświetlacza LCD 16*2               */
/*           - prosta gra zręcznościowa                          */
/* J.D. '2003                                                    */
/*****************************************************************/
#include <io.h>
#include <interrupt.h>
#include <signal.h>

#define lcd_rs 2      //definicja bitu portu dla linii RS
#define lcd_e 3       //definicja bitu portu dla linii E
#define tau0 247      //stała czasowa timera0
#define vliczt0 113   //stała wpisywana do licznika wejść do
                      //do przerwania timera0
#define vlkursor 10   //wartość wpisywana do zmiennej lkursor

//******************** zmienne globalne ********************
unsigned char liczt0;  //Licznik wejść do przerwania timera0.
                       //Klawisz jest badany, gdy liczt0=0
unsigned char lkursor; //licznik wejść do przerwania timera0 dla
                       //ustawienia flagi zmiany położenia "lisa"
volatile unsigned char kodklaw;  //kod naciśniętego klawisza;
volatile unsigned char fklaw;    //flaga wykrycia naciśnięcia klawisza;
volatile unsigned char fkursor;  //flaga zmiany położenia "lisa"
unsigned char wiersz=0;          //pozycja umieszczenia znaku na LCD
unsigned char kolumna=0;         //pozycja umieszczenia znaku na LCD

void czekaj(unsigned long zt) //procedura wytracania czasu
{
#define tau 10.38      //przybliżony przelicznik argumentu na ms
 unsigned char zt1;

 for(;zt>0;zt--)
 {
  for(zt1=255;zt1!=0;zt1--);
 }
}

SIGNAL (SIG_OVERFLOW0)  //obsługa przerwania od przepełnienia timera0
{
 unsigned char kkolumna,kwiersz; //zmienne pomocnicze
 TCNT0=tau0;           //odśwież stałą czasową w TCNT0

 if(--liczt0==0)       //czy już czytać klawisze?
 {                     //tak
  for(kkolumna=0xfe;kkolumna!=0xfb;kkolumna=(kkolumna<<1)+1)
  {
   PORTD=(PORTD|0x03)&(kkolumna);    //podaj "0" na linię sterującą aktywną kolumną
   kwiersz=PIND&0x0f;  //pozostaw tylko linie obsługujące klawiaturę
   if((kwiersz&0x0c)^0x0c)    //czy jest odpowiedź na jakiejś linii wierszy?
   {                          //tak, ustaw flagę gotowości klawiatury
    fklaw=1;
    break;                    //i zakończ przepatrywanie
   }
  }
  if(fklaw)
```

```
  {
   kodklaw=kwiersz;    //jeśli wykryto wciśnięcie klawisza, podaj jego kod
  }
  else
  {
   kodklaw=0xff;       //jeśli nie, ustaw kod neutralny
  }
  liczt0=vliczt0;      //odśwież stan liczt0
  if(--lkursor==0)     //czy można ustawić flagę zmiany "lisa"?
  {
   fkursor=1;          //tak (minęło ok. 300ms)
   lkursor=vlkursor;   //odśwież stan lkursor
  }
 }
}

void piszilcd(unsigned char instr) //zapisz instrukcję sterującą do LCD
{
 cbi(PORTB,lcd_rs);
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|(instr&0xf0);  //przygotuj starszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);                 //impuls strobujący
 czekaj(10L);                      //czekaj na gotowość LCD ok. 100us
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|((instr&0x0f)<<4);  //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);                 //impuls strobujący
 czekaj(10L);                      //czekaj na gotowość LCD ok. 100us
}

void piszdlcd(char dana)           //zapisz daną do LCD
{
 sbi(PORTB,lcd_rs);
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|(dana&0xf0);   //przygotuj starszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);                 //impuls strobujący
 czekaj(10L);                      //czekaj na gotowość LCD
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|((dana&0x0f)<<4); //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);                 //impuls strobujący
 czekaj(10L);                      //czekaj na gotowość LCD
}

void czysclcd(void)                //czyść ekran
{
 piszilcd(0x01);      //polecenie czyszczenia ekranu dla kontrolera LCD
 czekaj(1.64*tau);    //rozkaz 0x01 wykonuje się 1.64ms
```

```
 wiersz=0;
 kolumna=0;
}

void lcdxy(unsigned char w, unsigned char k)    //ustaw współrzędne kursora
{
 piszilcd((w*0x40+k)|0x80);  //standardowy rozkaz sterownika LCD
                        //ustawiający kursor w określonych współrzędnych
}

void piszznak(char znak)    //procedura umieszcza znak na wyświetlaczu
{
 piszdlcd(znak);            //wyświetl znak na LCD
}

unsigned char incwiersz(unsigned char w)  //inkrementuj wiersz LCD
{
 return w==1?0:1;
}

unsigned char decwiersz(unsigned char w)  //dekrementuj wiersz LCD
{
 return w==0?1:0;
}

unsigned char inckolumna(unsigned char k) //inkrementuj kolumnę LCD
{
 return k==15?0:++k;
}

unsigned char deckolumna(unsigned char k) //dekrementuj kolumnę LCD
{
 return k==0?15:--k;
}

int main(void)         //program główny
{
 unsigned char r,i;    //zmienne pomocnicze
 unsigned char xg=0;   //kolumna wyświetlania kursora gracza
 unsigned char yg=0;   //wiersz wyświetlania kursora gracza

 liczt0=vliczt0;
 lkursor=vlkursor;
 PORTD=0xff;           //z podciąganiem
 PORTB=0x03;           //z podciąganiem wejść
 DDRD=0x03;            //PD0-PD1 wy, pozostałe wejścia
 DDRB=0xff;            //PORTB - wy
 TIMSK=1<<TOIE0;       //zezwolenie na przerwania od TC0
 TCNT0=tau0;           //wpisz stałą czasowa do TCNT0
 TCCR0=4;              //preskaler XTAL/256,
                  //kwant mierzonego czasu = 32<m>s
 fklaw=0;
 //>>>>>>>>>>>>>>>>>>   inicjacja wyświetlacza LCD   <<<<<<<<<<<<<<<<<<<<<
 czekaj(45*tau);       //opóźnienie ok. 45ms dla ustabilizowania się napięcia
                   //zasilania LCD (katalogowo min. 15 ms)
                   //lcd_rs już wcześniej było ustawione w stanie "0"
 for(i=0;i<3;i++)      //3-krotne wysłanie 3-
 {
```

```
 sbi(PORTB,lcd_e);
 PORTB=(PORTB&0x0f)|0x30; //wyślij 3- do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 cbi(PORTB,lcd_e);
 czekaj(5*tau);      //ok. 5ms
}
sbi(PORTB,lcd_e);
PORTB=(PORTB&0x0f)|0x20;  //wyślij 2- do LCD
asm("nop");              //wymagane wydłużenie impulsu
asm("nop");
asm("nop");
cbi(PORTB,lcd_e);     //impuls strobujący
czekaj(10L);
piszilcd(0x28);       //interfejs 4-bitowy, 2 linie, znak 5x7
piszilcd(0x08);       //wyłącz LCD, wyłącz kursor, wyłącz mruganie
piszilcd(0x01);       //czyść LCD
czekaj(1.64*tau);     //wymagane dla instrukcji czyszczenia ekranu opóźnienie
piszilcd(0x06);       //bez przesuwania w prawo
piszilcd(0x0c);       //włącz LCD, bez kursora, bez mrugania
sei();               //odblokuj globalne przerwania
i=0;

piszznak('S');        //wyświetl napis "Start"
piszznak('t');
piszznak('a');
piszznak('r');
piszznak('t');

do
{
 i++;                //zmienna "i" będzie wykorzystana później do zainicjowania
                     //generatora pseudolosowego
}while(!fklaw);       //naciśnięcie klawisza zapewnia losowy start generatora
srand(i);             //inicjuj generator liczb pseudolosowych
czysclcd();           //czyść ekran
piszznak('+');        //umieść znak gracza na ekranie

while(1)              //główna pętla programu
{
 if(fkursor)          //czy można przemieścić "lisa"?
 {
  lcdxy(wiersz,kolumna);  //tak...
  if((wiersz==yg)&&(kolumna==xg))
  {
   piszznak('+');      //wymaż starą pozycję pozostawiając gracza
  }
  else
  {
   piszznak(' ');      //wymaż starą pozycję
  }
  r=rand();           //losuj nowe położenie
  if((r<37)&&(r>18))  //wylicz nową pozycję
  {
   wiersz=incwiersz(wiersz);
  }
  if(r<19)
```

```
 {
  wiersz=decwiersz(wiersz);
 }
 if(((r>12)&&(r<19))||((r>30)&&(r<147)))
 {
  kolumna=inckolumna(kolumna);
 }
 if((r<7)||((r<25)&&(r>18))||(r>146))
 {
  kolumna=deckolumna(kolumna);
 }
 lcdxy(wiersz,kolumna);     //ustaw nowe współrzędne
 piszznak('.');      //umieść "lisa" w nowym położeniu
 fkursor=0;
}

if(fklaw)
{                  //wykryto naciśnięcie klawisza
 fklaw=0;
 lcdxy(yg,xg);       //ustaw współrzędne znaku gracza...
 piszznak(' ');      //... i wymaż go
 switch (kodklaw)    //reakcja na klawisz
 {
  case 0x09:  xg=inckolumna(xg);      //SW4 - w prawo z zawijaniem
            break;
  case 0x05:  xg=deckolumna(xg);      //SW3 - w lewo z zawijaniem
            break;
  case 0x06:  yg=1;                   //SW2 - na dół bez zawijania
             break;
  case 0x0a:  yg=0;                   //SW1 - do góry bez zawijania
 }
 lcdxy(yg,xg);       //ustaw nowe współrzędne znaku gracza
 if((wiersz==yg)&&(kolumna==xg))
 {
  piszznak('*');     //znak, jeśli trafiony
  czekaj(200);        //przytrzymaj chwilę na ekranie
 }
 else
 {
  piszznak('+');     //znak jeśli "pudło"
 }
 }
 }
}
```

```
/*****************************************************************/
/* Ćwiczenie 6 - Zastosowanie komparatora analogowego do budowy */
/*              przetwornika analogowo-cyfrowego.                */
/*              Wyzwalanie funkcji przechwytywania timera1       */
/*              za pomocą komparatora.                           */
/*              Przerwanie od przechwytywania.                   */
/*              Obsługa wewnętrznej pamięci EEPROM.              */
/* J.D. '2003                                                    */
/*****************************************************************/
#include <io.h>
#include <interrupt.h>
#include <signal.h>
#include <eeprom.h>

//********************  zmienne globalne  *******************
unsigned char liczt0;
volatile unsigned char pomiar;    //flaga dokonania pomiaru
union{
     unsigned int wspkal;        //współczynnik kalibracji
     unsigned char wspkalb[2];
     }uwspkal;

void czekaj(unsigned long zt) //procedura wytracania czasu
{
 #define tau 10.38
 unsigned char zt1;
 for(;zt>0;zt--)
 {
  for(zt1=255;zt1!=0;zt1--);
 }
}

SIGNAL (SIG_INPUT_CAPTURE1)  //obsługa przerwania od przechwycenia
{
 union{
      unsigned int czas;
      unsigned char czasb[2];
     }uczas;
 unsigned char czas8;

 uczas.czasb[0]=ICR1L;       //zatrzaśnij rejestry przechwytywania
 uczas.czasb[1]=ICR1H;
 czas8=~((uczas.czas/uwspkal.wspkal)<<2);   //normalizacja wyniku
               //do postaci 6-bitowej liczby binarnej przesuniętej o 2 bity
               //w lewo (PORTB1 i 0 są wykorzystywane przez komparator
               //analogowy)
 PORTB=(PORTB&0x03)|czas8;   //wyświetl wynik na LEDach
 pomiar=1;                  //pomiar dokonany (zapal flagę)
 sbi(PORTD,4);              //zacznij rozładowywać kondensator pomiarowy
}

int main(void)
{
 DDRD=0x13;       //PORTD we oprócz PD4, PD1 i PD0 - wy
 PORTD=0xff;      //z podciąganiem
 PORTB=0x01;      //PB0 z podciąganiem
 DDRB=0xfc;       //PORTB7-2 - wy, PORTB1-0 - we
 TCCR1A=0;        //funkcje porównania i PWM wyłączone
```

```
TCCR1B=0x41;     //preskaler XTAL/1 dla TC1, przechwytywanie na
                 //narastającym zboczu
TIMSK=0x08;      //zezwolenie na przerwania od przechwytywania
ACSR=1<<ACIC;    //zezwolenie na wyzwalanie przechwytywania komparatorem
czekaj(10*tau);
TIFR=0xff;       //kasuj przerwania od timerów

if(bit_is_clear(PIND,1))
{           //wciśnięty SW4 - kalibracja
 do
 {
  cbi(PORTD,4);        //ładuj kondensator pomiarowy
  TCNT1H=0;      //zeruj licznik 1 pomiar czasu ładowania
  TCNT1L=0;
  sbi(TIFR,ICF1);
  while(bit_is_clear(TIFR,ICF1));    //czekaj aż napięcie mierzone zrówna się z napięciem
                                     //wejściowym
  uwspkal.wspkal=(ICR1L+256*ICR1H);
  sbi(PORTD,4);
  czekaj(1*tau);          //opóźnienie związane z częstotliwością odświeżania
 }while(bit_is_set(PIND,0));
 uwspkal.wspkal/=64;
 eeprom_wb(1,uwspkal.wspkalb[0]);  //zapisz współczynnik kalibracji
 eeprom_wb(2,uwspkal.wspkalb[1]);  //do pamięci EEPROM
 sbi(TIFR,ICF1);
}
else
{
 uwspkal.wspkal=eeprom_rw(1);  //odczytaj współczynnik kalibracji z EEPROM-u
}
sei();               //odblokuj globalne przerwania
while(1)             //główna pętla pomiarowa
{
 cbi(PORTD,4);        //ładuj kondensator pomiarowy
 TCNT1H=0;            //zeruj licznik 1 - pomiar czasu ładowania
 TCNT1L=0;
 pomiar=0;
 while(pomiar==0);    //czekaj aż napięcie mierzone zrówna się z napięciem
                  //na kondensatorze pomiarowym
 czekaj(23*tau);      //opóźnienie związane z częstotliwością odświeżania
                  //wskaźnika LED
 }
}
```

```
/*****************************************************************/
/* Ćwiczenie 7 - Regulacja obrotów silnika DC                    */
/*              Modulacja PWM przy użyciu timera1                */
/* J.D. '2003                                                    */
/*****************************************************************/
#include <io.h>

void czekaj(unsigned long zt) //procedura wytracania czasu
{
 #define tau 10.38
 unsigned char zt1;
```

```
 for(;zt>0;zt--)
 {
  for(zt1=255;zt1!=0;zt1--);
 }
}

int main( void )
{
 unsigned char licznikkl=0;  //zmienna wykorzystywana do pomiaru czasu
                             //naciśnięcia przycisków
 char przyrost=1;  //przyrost zmiany współczynnika wypełnienia sygnału PWM
 union
     {
      unsigned int pwm;
      unsigned char pwmc[2];
     }volatile upwm;        //aktualny współczynnik wypełnienia sygn. PWM

 DDRB=0x08;    //PB3 - wy (OC1 - wyjście PWM), pozostałe we
 PORTB=0;      //bez podciągania
 DDRD=0xfc;    //PD1 i PD0 - we (obsługa klawiszy SW1 i SW4), pozostałe wy
 PORTD=0x03;   //wejścia z podciąganiem (potrzebne dla klawiatury)
 TCCR1A=0x83;  //PWM 10 bitowy
               //zerowanie OC1 po spełnieniu warunku równości podczas liczenia
               //w górę,
               //ustawiane podczas liczenia w dół
 TCCR1B=0x03;  //preskaler=3, co przy 10-bit PWM daje Fwy=ok. 61Hz @8MHz
 TCNT1L=0x00;  //wstępne ustawienie licznika1
 TCNT1H=0x00;
 upwm.pwm=0x3ff;   //początkowo silnik włączony, wartość TOP odpowiada wysokiemu
                   //poziomowi na wyjściu OC1 (PB3)
 while(1)          //główna pętla programu
 {
  if(bit_is_clear(PIND,1))   //czy wciśnięto SW4
  {
   upwm.pwm+=przyrost;       //zwiększ pwm
   if(upwm.pwm>0x3ff)
   {
    upwm.pwm=0x3ff;          //jeśli przekroczono wartość TOP, to ustaw TOP
   }
   czekaj(150*tau);          //eliminacja drgań i powtórnej interpretacji
                             //naciśnięcia przycisku
   licznikkl++;              //mierz długość naciśnięcia przycisku
  }
  else
  {
   if(bit_is_clear(PIND,0))  //czy wciśnięto SW1
   {
    upwm.pwm-=przyrost;      //zmniejsz pwm
    if(upwm.pwm>0x3ff)
    {
     upwm.pwm=0;             //jeśli przekroczono wartość zero to ustaw zero
    }
    czekaj(150*tau);         //eliminacja drgań i powtórnej interpretacji
                             //naciśnięcia przycisku
    licznikkl++;             //mierz długość naciśnięcia przycisku
   }
   else
   {
```

```
   licznikkl=0;            //zeruj licznik pomiaru czasu naciśnięcia klawisza
                          //ponieważ wszystkie przyciski są zwolnione
   przyrost=1;             //ustaw początkową wartość przyrostu dla pwm
  }
 }
 if(licznikkl>6)
 {
  przyrost=+16;           //wykryto długie naciśnięcie przycisków,
                          //zwiększ krok regulacji
  licznikkl=6;
 }
 OCR1H=upwm.pwmc[1];      //wpisz aktualnie ustawiony współczynnik do rejestrów
 OCR1L=upwm.pwmc[0];      //OCR1 timera1
}
}
```

```
/*****************************************************************/
/* Ćwiczenie 8 - Zdalna regulacja obrotów silnika DC z komputera PC */
/*            Modulacja PWM przy użyciu timera1                  */
/*            Wykorzystanie UART-a do transmisji z komputerem    */
/* J.D. '2003                                                    */
/*****************************************************************/

#include <io.h>
#include <progmem.h>
#include <stdlib.h>
#include <interrupt.h>
#include <signal.h>

#define FCPU     8000000    //częstotliwość oscylatora CPU
#define VUART    38400      //prędkość transmisji [bit/s]
#define VUBRR    FCPU/(VUART*16)-1    //wpis do UBRR dla VUART

unsigned char romram;       //romram=1 => dane z pamięci programu
                           //romram=0 => dane z RAM-u
char *pfifosio;             //wskaźnik na kolejkę UART-u
unsigned char volatile fodbznak=0;    //flaga: "odebrano znak"
char komenda;               //odebrana komenda z PC-ta
char *fifosio[];            //wskaźnik na kolejkę UART-u


SIGNAL(SIG_UART_RECV)       //procedura obsługi odbiornika UART-u
{
 komenda=UDR;               //zapamiętaj odebraną komendę
 fodbznak=1;                //ustaw flagę odebrania znaku
}

SIGNAL(SIG_UART_TRANS)      //procedura obsługi nadajnika UART
{                          //wywoływana po wysłaniu znaku
 char znak;

 if(romram)                 //skąd pobierać dane?
 {
  znak=PRG_RDB(pfifosio++); //pobierz daną z pamięci programu
```

```
 }
 else
 {
  znak=*pfifosio++;        //pobierz dane z pamięci RAM
 }
 if(znak!=0)              //czy koniec pobierania danych?
 {
  UDR=znak;               //nie, wyślij znak pobrany z kolejki
 }
 else
 {
  cbi(UCR,TXEN);          //tak, wyłącz nadajnik
 }
}

void czekaj(unsigned long zt) //procedura wytracania czasu
{
 #define tau 10.38
 unsigned char zt1;
 for(;zt>0;zt--)
 {
  for(zt1=255;zt1!=0;zt1--);
 }
}

void wyslijtekstROM(char *tekst)  //wysyłanie danych z pamięci programu
{
 romram=1;                //dane będą z pamięci programu
 pfifosio=tekst;          //ustaw wskaźnik na dane do wysłania
 sbi(UCR,TXEN);           //włącz nadajnik
 UDR=PRG_RDB(pfifosio++);  //wyślij pierwszy znak, pozostałe będą pobierane
                          //w procedurze obsługi przerwania TXC
}

void wyslijtekst(char *tekst)     //wysyłanie danych z pamięci programu
{
 romram=0;                //dane będą z pamięci danych
 pfifosio=tekst;          //ustaw wskaźnik na dane do wysłania
 sbi(UCR,TXEN);           //włącz nadajnik
 UDR=*pfifosio++;         //wyślij pierwszy znak, pozostałe będą pobierane
                          //w procedurze obsługi przerwania TXC
}

int main(void)
{
 unsigned char i;
 unsigned char volatile licznikkl=0;  //zmienna wykorzystywana do pomiaru czasu
                          //naciśnięcia przycisków

 char volatile przyrost=1;  //przyrost zmiany współczynnika wypełnienia sygnału PWM
 //tablica komunikatów do wysłania
 char *info[7]={
           PSTR("\n\rRegulator obrotów silnika DC\n\r"),
           PSTR(", - zmniejszanie obrotów\n\r"),
          PSTR(". - zwiększanie obrotów\n\r"),
          PSTR("0 - zatrzymanie silnika\n\r"),
          PSTR("1 - start z max. obrotami\n\r"),
          PSTR("N - podaj aktualne parametry sterownika\n\r\n"),
```

```
            PSTR("\n\rAktualne parametry PWM:")
            };

union                   //unia pozwala na bajtowy dostęp do zmiennej int
    {
     unsigned int pwm;
     unsigned char pwmc[2];
    }volatile upwm;       //aktualny współczynnik wypełnienia sygn. PWM

DDRB=0xff;    //PORTB - wy
PORTB=0xff;
DDRD=0x02;    //PD1 - wy (RXD), pozostałe we
PORTD=0x02;   //podciągania wejścia PD1 (RXD)
UBRR=VUBRR;   //ustaw prędkość transmisji
UCR=1<<RXCIE | 1<<TXCIE  | 1<<RXEN;   //zezwolenie na przerwania od
              //odbiornika i nadajnika, zezwolenie na odbiór i nadawanie
TCCR1A=0x83;      //PWM 10 bitowy
              //zerowanie OC1 po spełnieniu warunku równości podczas liczenia
              //w górę, ustawiane podczas liczenia w dół
TCCR1B=0x01;      //preskaler=3, co przy 10-bit PWM daje Fwy=ok. 61Hz @8MHz
TCNT1L=0x00;      //wstępne ustawienie licznika 1
TCNT1H=0x00;
upwm.pwm=0x3ff;   //początkowo silnik włączony, wartość TOP odpowiada wysokiemu
              //poziomowi na wyjściu OC1 (PB3)
OCR1H=upwm.pwmc[1];         //wpisz aktualnie ustawiony współczynnik do rejestrów
OCR1L=upwm.pwmc[0];         //OCR1 timera1
sei();          //włącz przerwania

for(i=0;i<5;i++) //wyślij winietkę
{
 wyslijtekstROM(info[i]);   //wysłanie pojedynczej linii tekstu
 while(bit_is_set(UCR,TXEN));    //trzeba zaczekać, aż zostanie wysłana do końca
}

while(1)        //główna pętla programu
{
 if(fodbznak)         //czy odebrano jakiś znak?
 {
  fodbznak=0;         //tak
  switch (komenda)     //interpretacja komendy i wykonanie odpowiedniej akcji
  {
   case '.':                //odebrano "." - zwiększ prędkość
            upwm.pwm+=przyrost;       //zwiększ PWM
            if(upwm.pwm>0x3ff)
            {
             upwm.pwm=0x3ff;    //jeśli przekroczono wartość TOP, to ustaw TOP
            }
            czekaj(150*tau);    //eliminacja powtórnej interpretacji
                        //naciśnięcia przycisku
            licznikkl++;              //mierz długość naciśnięcia przycisku
           break;
   case ',':                //odebrano "," - zmniejsz prędkość
            upwm.pwm-=przyrost;       //zmniejsz PWM
            if(upwm.pwm>0x3ff)
            {
             upwm.pwm=0;        //jeśli przekroczono wartość zero to ustaw zero
            }
            czekaj(150*tau);    //eliminacja powtórnej interpretacji
```

```
                        //naciśnięcia przycisku
            licznikkl++;        //mierz długość naciśnięcia przycisku
            break;
    case '0':               //odebrano "0" - zatrzymaj silnik
            upwm.pwm=0;     //silnik STOP
            break;
    case '1':               //odebrano "1" - ustaw max obroty silnika
            upwm.pwm=0x3ff;     //silnik na MAX
            break;
    case 'n':
    case 'N':
            wyslijtekstROM(info[6]);      //wysłanie pojedynczej linii tekstu
            while(bit_is_set(UCR,TXEN));  //trzeba zaczekać,
                                          //aż zostanie wysłana do końca
            wyslijtekst("0x");       //wyślij prefiks dla liczb heksadecymalnych
            while(bit_is_set(UCR,TXEN));  //trzeba zaczekać,
                                          //aż zostanie wysłana do końca
            itoa(upwm.pwm,fifosio,16);   //konwersja liczby int (hex)
                                          //na łańcuch znakowy
            wyslijtekst(fifosio);        //wyślij aktualną wartość PWM do PC-ta
            while(bit_is_set(UCR,TXEN));  //trzeba zaczekać,
                                          //aż zostanie wysłana do końca
            break;
   }
   if(licznikkl>6)
   {
    przyrost=+16;            //wykryto długie naciśnięcie klawisza,
                        //zwiększ krok regulacji
    licznikkl=6;             //dalej już nie zwiększaj kroku
    cbi(PORTB,1);            //zapal diodę LED2
   }
   OCR1H=upwm.pwmc[1];       //wpisz aktualnie ustawiony współczynnik do rejestrów
   OCR1L=upwm.pwmc[0];       //OCR1 timera1
  }
  else
  {                     //jeśli cisza na linii, ustaw parametry spoczynkowe
   licznikkl=0;
   przyrost=1;
   sbi(PORTB,1);            //zgaś diodę LED2
  }
 }
}



/*********************************************************************/
/* Ćwiczenie 9 - Obsługa interfejsu 1-wire - odczyt pastylki DS1990A */
/*            Obsługa wyświetlacza LCD 2x16                          */
/* J.D. '2003                                                        */
/*********************************************************************/

#include <io2313.h>
#include <progmem.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
// Poniższe definicje służą do realizacji wygodnego dostępu bitowego
typedef struct _bit_struct
{
 unsigned char bit0: 1;
 unsigned char bit1: 1;
 unsigned char bit2: 1;
 unsigned char bit3: 1;
 unsigned char bit4: 1;
 unsigned char bit5: 1;
 unsigned char bit6: 1;
 unsigned char bit7: 1;
}pole_bitowe;

#define DAJ_BIT(adr) (*((volatile pole_bitowe*) (adr)))
#define _PORTB 0x38
#define _PINB 0x36
#define _PORTD 0x32
#define _DDRD 0x31
#define _PIND 0x30
#define pastylka_we DAJ_BIT(_PIND).bit3
#define pastylka_wy DAJ_BIT(_DDRD).bit3
#define lcd_rs DAJ_BIT(_PORTB).bit2
#define lcd_e DAJ_BIT(_PORTB).bit3
#define led0 DAJ_BIT(_PORTB).bit0

#define stan_0 1   //definicja stanu niskiego na linii1-wire
                   //"1" oznacza przełączenie portu w tryb wyjściowy
                   //port jest wcześniej wysterowany w stan niski
#define stan_1 0   //definicja stanu wysokiego na linii1-wire
                   //"0" oznacza przełączenie portu w tryb wejściowy
                   //stan wysoki jest wymuszany przez zewnętrzny
                   //rezystor podciągający
#define CR 0x0a    //definicja znaku CR (przecie do nowej linii)

char buflcd[4];    //roboczy bufor wyświetlacza LCD
char *pbuflcd;     //wskaźnik na bufor wyświetlacza
unsigned char bufor1w[8];    //bufor interfejsu 1-wire (dane z pastylki)
unsigned char *pbufor1w;     //wskaźnik na bufor danych z pastylki
unsigned char wiersz=0;      //pozycja umieszczenia znaku na LCD
unsigned char kolumna=0;     //pozycja umieszczenia znaku na LCD

void czekaj(unsigned long zt) //procedura wytracania czasu
{
 #define tau 10.38
 unsigned char zt1;
 for(;zt>0;zt--)
 {
  for(zt1=255;zt1!=0;zt1--);
 }
}

//>>>>>>>>>>>>>    Procedury obsługi wyświetlacza LCD   <<<<<<<<<<<<<<<<<<
void piszilcd(unsigned char instr)     //zapisz instrukcję sterującą do LCD
{
 lcd_rs=0;
 lcd_e=1;
 PORTB=(PORTB&0x0f)|(instr&0xf0);  //przygotuj starszy półbajt do LCD
 asm("nop");
```

```
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD ok. 100us
 lcd_e=1;
 PORTB=(PORTB&0x0f)|((instr&0x0f)<<4); //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD ok. 100us
}

void piszdlcd(char dana)     //zapisz daną do LCD
{
 lcd_rs=1;
 lcd_e=1;
 PORTB=(PORTB&0x0f)|(dana&0xf0);  //przygotuj starszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD
 lcd_e=1;
 PORTB=(PORTB&0x0f)|((dana&0x0f)<<4);  //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD
}

void czysclcd(void)          //czyść ekran
{
 piszilcd(0x01);   //polecenie czyszczenia ekranu dla kontrolera LCD
 czekaj(1.64*tau); //rozkaz 0x01 wykonuje się 1.64ms
 wiersz=0;
 kolumna=0;
}

void lcdxy(unsigned char w, unsigned char k)     //ustaw współrzędne kursora
{
 piszilcd((w*0x40+k)|0x80);  //standardowy rozkaz sterownika LCD
}                            //ustawiający kursor w określonych współrzędnych

void piszznak(char znak)     //procedura umieszcza znak na wyświetlaczu
{
 piszdlcd(znak);             //wyświetl znak na LCD
}

void pisztekst(char *tekst)  //pisz tekst na LCD wskazywany pointerem *tekst
{
 char zn;
 char nr=0;

 while(1)
 {
  zn=PRG_RDB(&tekst[nr++]);  //pobranie znaku z pamięci programu
```

```
 if(zn!=0)                //czy nie ma końca tekstu?
 {
  if(zn==CR)               //czy znak nowej linii
  {
   wiersz==1?wiersz=0:++wiersz;       //przejdź do nowej linii
   kolumna=0;
   lcdxy(wiersz,kolumna);   //ustaw obowiązujące po zmianie współrzędne na LCD
  }
  else
  {
   piszdlcd(zn);            //umieść pojedynczy znak tekstu na LCD
  }
 }
 else
 {
  break;                   //zakończ pętlę, jeśli koniec tekstu
 }
}

//>>>>>>>>>>>>>    Procedury obsługi interfejsu 1-Wire   <<<<<<<<<<<<<<<<<<
void czekaj_1w(unsigned char tau1w)    //pętla opóźnienia dla 1-wire
{                    //opóźnienie jest równe t=(5+5*tau1w)*T
 do                   //T - cykl zegarowy MCU, tau1w - parametr procedury
 {
  asm("nop");          //wstawka asemblerowa - rozkaz NOP mikrokontrolera
 }while(--tau1w!=0);
}

unsigned char slot1w_czyt(void)  //slot odczytu pastylki
{
 unsigned char bit1w;

 pastylka_wy=stan_0;         //inicjuj slot czytania
 czekaj_1w(1);              //odczekaj tsu
 pastylka_wy=stan_1;         //zwolnij linię
 czekaj_1w(27);              //ok. 14us
 bit1w=pastylka_we==1?1:0;
 czekaj_1w(118);             //ok. 60us
 return bit1w;
}

void slot1w_zap(unsigned char znak)       //slot zapisu do pastylki
{
 if(znak)
 {                        //slot "1"
  pastylka_wy=stan_0;        //inicjuj slot
  czekaj_1w(20);            //ok. 11us
  pastylka_wy=stan_1;        //podciąganie do "1" rezystorem zewnętrznym
  czekaj_1w(119);           //60us do końca slotu
 }
 else
 {                        //slot "0"
  pastylka_wy=stan_0;        //inicjuj slot
  czekaj_1w(139);           //ok. 70us
  pastylka_wy=stan_1;        //podciąganie do "1" rezystorem zewnętrznym
 }
}
```

```
void zapisz1w(unsigned char rozkaz)    //transmisja 8-bitowego rozkazu do pastylki
{
 unsigned char i;

 for(i=0;i<8;i++)
 {
  slot1w_zap(rozkaz&0x01); //wyślij bit do pastylki
  rozkaz>>=1;
 }
}

void czytaj1w(void)          //odczyt bajtu z pastylki
{
 unsigned char i,j;
 unsigned char dana;

 pbufor1w=&bufor1w[0];  //dane będą umieszczone w buforze "bufor1w"
 for(i=0;i<8;i++)       //czytaj "family code" (1bajt)
 {                      //i "registration number" (6 bajtów)
  dana=0;               //wstępne zerowanie danej
  for(j=0x01;j!=0;j<<=1)     //zmienna sterująca pętli wskazuje jednocześnie
  {                          //aktualnie zapisywany bit
   dana|=slot1w_czyt()?j:dana;   //czytaj kolejne bity (są one sumowane logicznie
  }                          //z odczytanymi wcześniej
  *pbufor1w++=dana;          //po skompletowaniu zapisz odebrany bajt do bufora
 }
}

void licz_CRC(char bajt,unsigned char *CRC)      //procedura wyliczania CRC
{                //wielomian generujący jest równy: x^8 + x^5 + x^4 + 1
 unsigned char zp1,zp2,i;    //zmienne pomocnicze

 zp1=bajt;
 for(i=0;i<8;i++)
 {
  bajt^=*CRC;           //wskaźnik *CRC wyznacza aktualnie wyliczony CRC
  zp2=bajt&0x01;        //wydzielenie bitu do obliczeń
  bajt=*CRC;
  if(zp2)
  {
   bajt^=0x18;
  }
  bajt=((unsigned char)(bajt)>>1)+0x80*zp2; //konwersja uch jest potrzebna do
                                            //prawidłowego wykonania przesunięcia
  *CRC=bajt;
  zp1=(bajt=zp1>>1);
 }
}

int main(void)     //program główny
{
 unsigned char i,zp;

 //tablica komunikatów do wyświetlenia
 char *info[5]={
            PSTR("Wykryto pastylke"),
            PSTR("Wykryto zwarcie "),
```

```
             PSTR("Brak pastylki   "),
            PSTR("                "),
            PSTR("Blad CRC        "),
            };

DDRB=0xff;       //PORTB - wy
DDRD=0;          //PORTD - we
PORTD=0;         //bez podciągania
led0=1;          //gaś LED-a

//>>>>>>>>>>>>>>>>>>    Inicjowanie wyświetlacza LCD
<<<<<<<<<<<<<<<<<<<<<<<<
lcd_rs=0;
czekaj(45*tau);       //opóźnienie ok. 45ms dla ustabilizowania się napięcia
                  //zasilania LCD (katalogowo min. 15 ms)
for(i=0;i<3;i++)      //3-krotne wysłanie 3-
{
 lcd_e=1;
 PORTB=(PORTB&0x0f)|0x30;   //wyślij 3- do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;
 czekaj(5*tau);       //ok. 5ms
}
lcd_e=1;
PORTB=(PORTB&0x0f)|0x20;    //wyślij 2- do LCD
asm("nop");               //wymagane wydłużenie impulsu
asm("nop");
asm("nop");
lcd_e=0;              //impuls strobujący
czekaj(10L);
piszilcd(0x28);       //interfejs 4-bitowy, 2 linie, znak 5x7
piszilcd(0x08);       //wyłącz LCD, wyłącz kursor, wyłącz mruganie
piszilcd(0x01);       //czyść LCD
czekaj(1.64*tau);     //wymagane dla instrukcji czyszczenia ekranu opóźnienie
piszilcd(0x06);       //bez przesuwania w prawo
piszilcd(0x0c);       //włącz LCD, bez kursora, bez mrugania

while(1)              //główna pętla programu
{
 zp=0;
 pastylka_wy=stan_0;  //sekwencja inicjująca pastylkę
 czekaj(5);           //impuls inicjujący "0" ok. 480us
 pastylka_wy=stan_1;  //zwolnij linię
 czekaj_1w(128);      //po odczekaniu ok. 65us czekaj na impuls obecności
 if(pastylka_we==0)
 {
  zp++;               //zapamiętaj fakt wykrycia impulsu obecności pastylki
 }
 czekaj(4);
 czekaj_1w(56);       //czekaj ok. 416us do zakończenia slotu inicjującego
 if(pastylka_we==0)
 {
  zp++;          //jeśli pozostaje w stanie niskim, to oznacza zwykłe zwarcie
 }
 if(zp==1)       //reakcja na rozpoznane sytuacje
 {              //jeśli zp=1, to oznacza że pastylka zgłosiła się
```

```
  zapisz1w(0x33); //wyślij do pastylki kod rozkazu przesłania numeru seryjnego
  czytaj1w();     //czytaj numer seryjny pastylki (+ kod rodziny + CRC)
  zp=0;           //w zp będzie liczone CRC - wstępne zerowanie
  for(i=0;i<8;i++)
  {
   licz_CRC(bufor1w[i],&zp);     //licz CRC ze wszystkich bajtów odebranych
  }
  lcdxy(0,0);
  if(zp==0)
  {               //zerowa wartość CRC oznacza prawidłowość danych
   led0=0;        //zapal LED-a
   pisztekst(info[0]);      //wyświetl komunikat na LCD o rozpoznaniu pastylki
   lcdxy(1,0);
   pbufor1w=&bufor1w[7];
   for(i=6;i!=0xff;i--)     //wyświetlenie danych z pastylki
   {
    utoa((unsigned char)bufor1w[i],buflcd,16); //przepisz bufor1w do
                //bufora wyświetlacza z jednoczesną konwersją na ASCII
    pbuflcd=&buflcd[0];     //ustaw wskaźnik bufora LCD na początek
    if(strlen(buflcd)<2)    //jeśli odebrano pojedynczą cyfrę, trzeba dopisać 0
    {
     piszznak('0');         //wyświetlenie zera wiodącego
    }
    while(*pbuflcd)         //wyświetl zawartość bufora LCD
    {
     piszznak(toupper(*pbuflcd++));  //wyświetl dane z konwersją na duże litery
    }
   }
  }
  else
  {
   pisztekst(info[4]);      //komunikat o błędzie obliczenia CRC
  }
  czekaj(3000*tau);         //przytrzymaj wynik na wyświetlaczu przez ok. 3s
 }
 else
 {
  led0=1;                   //gaś LED-a
  lcdxy(1,0);
  pisztekst(info[3]);       //wyczyść dolną linię wyświetlacza
  lcdxy(0,0);
  if(zp==2)
  {                         //to było tylko zwarcie
   pisztekst(info[1]);      //wyświetl komunikat o zwarciu
  }
  else
  {
   pisztekst(info[2]);      //wyświetl komunikat o braku pastylki
  }
 }
}
```

```
/*********************************************************************/
/* Ćwiczenie 10 - Obsługa interfejsu I2C                             */
/*              układ RTC (zegar czasu rzeczywistego) - PCF8583      */
/*              Obsługa przerwania zewnętrznego INT0                 */
/*              Obsługa wyświetlacza LCD 2x16                        */
/* J.D. '2003                                                        */
/*********************************************************************/

#include <io2313.h>
#include <progmem.h>
#include <interrupt.h>
#include <signal.h>

// Poniższe definicje służą do realizacji wygodnego dostępu bitowego
typedef struct _bit_struct
{
 unsigned char bit0: 1;
 unsigned char bit1: 1;
 unsigned char bit2: 1;
 unsigned char bit3: 1;
 unsigned char bit4: 1;
 unsigned char bit5: 1;
 unsigned char bit6: 1;
 unsigned char bit7: 1;
}pole_bitowe;

#define DAJ_BIT(adr) (*((volatile pole_bitowe*) (adr)))
#define _PORTB 0x38
#define _PINB 0x36
#define _PORTD 0x32
#define _DDRD 0x31
#define _PIND 0x30
#define sda_we DAJ_BIT(_PIND).bit5
#define sda_wy DAJ_BIT(_DDRD).bit5
#define scl_wy DAJ_BIT(_DDRD).bit6
#define scl_we DAJ_BIT(_PIND).bit6
#define lcd_rs DAJ_BIT(_PORTB).bit2
#define lcd_e DAJ_BIT(_PORTB).bit3
#define sw4 DAJ_BIT(_PIND).bit1
#define sw1 DAJ_BIT(_PIND).bit0

#define stan_0 1   //definicja stanu niskiego na liniach I2C
                   //"1" oznacza przełączenie portu w tryb wyjściowy
                   //port jest wcześniej wysterowany w stan niski
#define stan_1 0   //definicja stanu wysokiego na liniach I2C
                   //"0" oznacza przełączenie portu w tryb wejściowy
                   //stan wysoki jest wymuszany przez zewnętrzny
                   //rezystor podciągający

#define rtc 0xa0   //7-bitowy adres bazowy zegara RTC (PCF8583) = 1010000
                   //przesunięty na bity od 7 do 1 i uzupełniony zerem
#define CR 0x0a    //definicja znaku CR (przecie do nowej linii)

char buflcd[4];    //roboczy bufor wyświetlacza LCD
char *pbuflcd;     //wskaźnik na bufor wyświetlacza
unsigned char bufi2c[9];    //bufor interfejsu I2C (dane z/do RTC)
unsigned char *pbufi2c;     //wskaźnik na bufor danych z RTC
unsigned char wiersz=0;     //pozycja umieszczenia znaku na LCD
```

```
unsigned char kolumna=0;     //pozycja umieszczenia znaku na LCD
unsigned char godz=0,min=0,sek=0,dzien=1,mies=1;
unsigned char fzegar;        //flaga odświeżenia stanu zegara na LCD

void czeka(unsigned long zt)     //procedura wytracania czasu
{
 #define tau 10.38           //przybliżony przelicznik argumentu na ms
 unsigned char zt1;
 for(;zt>0;zt--)
 {
  for(zt1=255;zt1!=0;zt1--);
 }
}

//>>>>>>>>>>>>>    Procedury obsługi wyświetlacza LCD   <<<<<<<<<<<<<<<<<<
void piszilcd(unsigned char instr)     //zapisz instrukcję sterującą do LCD
{
 lcd_rs=0;
 lcd_e=1;
 PORTB=(PORTB&0x0f)|(instr&0xf0);      //przygotuj starszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD ok. 100us
 lcd_e=1;
 PORTB=(PORTB&0x0f)|((instr&0x0f)<<4); //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD ok. 100us
}

void piszdlcd(char dana)     //zapisz daną do LCD
{
 lcd_rs=1;
 lcd_e=1;
 PORTB=(PORTB&0x0f)|(dana&0xf0);  //przygotuj starszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD
 lcd_e=1;
 PORTB=(PORTB&0x0f)|((dana&0x0f)<<4);  //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD
}

void czysclcd(void)          //czyść ekran
{
 piszilcd(0x01);   //polecenie czyszczenia ekranu dla kontrolera LCD
 czekaj(1.64*tau); //rozkaz 0x01 wykonuje się 1.64ms
 wiersz=0;
```

```
 kolumna=0;
}

void lcdxy(unsigned char w, unsigned char k)     //ustaw współrzędne kursora
{
 piszilcd((w*0x40+k)|0x80);  //standardowy rozkaz sterownika LCD
}                            //ustawiający kursor w określonych współrzędnych

void piszznak(char znak)     //procedura umieszcza znak na wyświetlaczu
{
 piszdlcd(znak);             //wyświetl znak na LCD
}

void pisztekst(char *tekst)  //pisz tekst na LCD wskazywany pointerem *tekst
{
 char zn;
 char nr=0;

 while(1)
 {
  zn=PRG_RDB(&tekst[nr++]);  //pobranie znaku z pamięci programu
  if(zn!=0)                  //czy nie ma końca tekstu?
  {
   if(zn==CR)                //czy znak nowej linii
   {
    wiersz==1?wiersz=0:++wiersz;       //przejdź do nowej linii
    kolumna=0;
    lcdxy(wiersz,kolumna);   //ustaw obowiązujące po zmianie współrzędne na LCD
   }
   else
   {
    piszdlcd(zn);            //umieść pojedynczy znak tekstu na LCD
   }
  }
  else
  {
   break;                    //zakończ pętlę, jeśli koniec tekstu
  }
 }
}

void nalcd(unsigned char zn1, unsigned char zn2)
{                 //umieszczenie czasu na LCD z jednoczesną konwersją na ASCII
 piszznak(zn1+0x30);    //dodanie 0x30 realizuje prostą konwersję liczby na ASCII
 piszznak((zn2&0x0f)+0x30);
}

//>>>>>>>>>>>>>>>>      Procedury obsługi interfejsu I2C    <<<<<<<<<<<<<<<<<<<<<
void czekaj_i2c(unsigned char taui2c)  //pętla opóźnienia dla I2C
{                      //opóźnienie jest równe t=(5+5*taui2c)*T
 do                     //T - cykl zegarowy MCU, taui2c - parametr procedury
 {
  asm("nop");           //wstawka asemblerowa - rozkaz NOP mikrokontrolera
 }while(--taui2c!=0);
}
```

```
void bitstartu(void)   //bit startu na magistrali I2C
{                      //     SCL    SDA
 sda_wy=stan_1;        //                 |
 czekaj_i2c(10);       //                 |
 scl_wy=stan_1;        //           |     |
 czekaj_i2c(10);       //           |     |
 sda_wy=stan_0;        //           |    /
 czekaj_i2c(10);       //           |   |
 scl_wy=stan_0;        //          /    |
 czekaj_i2c(10);       //        |      |
}                      //        |      |

void bitstopu(void)
{                       //     SCL    SDA
 sda_wy=stan_0;         //    |       |
 czekaj_i2c(10);        //    |       |
 scl_wy=stan_1;         //     \      |
 czekaj_i2c(10);        //       |    |
 sda_wy=stan_1;         //       |     \
 czekaj_i2c(10);        //       |      |
}                       //       |      |

unsigned char zapiszB_i2c(unsigned char dana)
{                    //wysłanie pojedynczego bajtu do Slave'a I2C
 unsigned char i;

 for(i=0;i<8;i++)       //będzie 8 bitów
 {
  if(dana&0x80)         //badaj najstarszy bit wysyłanego znaku
  {
   sda_wy=stan_1;       //wyślij "1"
  }
  else
  {
   sda_wy=stan_0;       //wyślij "0"
  }
  czekaj_i2c(10);
  scl_wy=stan_1;
  czekaj_i2c(10);
  scl_wy=stan_0;
  czekaj_i2c(10);
  dana<<=1;         //przygotuj następny bit do wysłania
 }
 sda_wy=stan_1;    //zwolnij linię SDA dla Slave'a
 czekaj_i2c(10);
 scl_wy=stan_1;
 while(!scl_we);   //czekaj, aż Slave będzie gotowy do przyjęcia następnego bitu
 czekaj_i2c(10);
 i=sda_we;         //czytaj potwierdzenie ACK od Slave'a
 scl_wy=stan_0;
 czekaj_i2c(10);
 return i;         //zwróć bit potwierdzenia ACK
}

unsigned char czytajB_i2c(unsigned char ack)
{       //przyjęcie pojedynczego bajtu od Slave'a I2C
        //parametr ack=1 oznacza, że jest wysyłany ostatni bajt bloku,
        //nie należy więc wysyłać potwierdzenia
```

```
 unsigned char dana=0,i;

 for(i=0;i<8;i++)  //będzie 8 bitów
 {
  scl_wy=stan_0;
  czekaj_i2c(10);
  scl_wy=stan_1;
  czekaj_i2c(10);
  dana<<=1;            //przygotuj miejsce na kolejny odebrany bit
  dana|=sda_we==1?1:0;  //dopisz odebrany bit
 }
 scl_wy=stan_0;
 if(ack)          //czy wysyłać potwierdzenie?
 {
  sda_wy=stan_0;   //wyślij ACK
 }
 else
 {
  sda_wy=stan_1;   //brak potwierdzenia sygnalizuje zakończenie bloku
 }
 czekaj_i2c(10);
 scl_wy=stan_1;
 while(!scl_we);   //czekaj, aż Slave będzie gotowy do przyjęcia następnego bitu
 czekaj_i2c(10);
 sda_wy=stan_1;    //zwolnij linię SDA
 scl_wy=stan_0;
 return dana;
}

void bladi2c(void)
{                //reakcja na brak potwierdzenia ACK od Slave'a
 char *tekst[2]={
             PSTR("Blad I2C"),
             PSTR("        ")
             };
 lcdxy(0,0);
 pisztekst(tekst[0]);   //wyświetl na LCD komunikat o błędzie I2C
 czekaj(1500*tau);
 lcdxy(0,0);
 pisztekst(tekst[1]);   //wyczyść komunikat o błędzie
}

void doi2c(unsigned char adri2c,unsigned char adrdane,unsigned char lz)
{       //wysłanie bloku "lz" znaków do Slave'a I2C
        //"adri2c" - adres fizyczny Slave'a
        //"adrdane" - adres rejestru w Slave'ie, od którego będą zapisywane dane
 pbufi2c=&bufi2c;  //ustaw wskaźnik bufora i2c na jego początek
 bitstartu();      //wyślij bit startu (początek transmisji)
 if(zapiszB_i2c(adri2c))     //wyślij adres Slave'a, to informacja dla pozostałych
                             //układów Slave, że dane nie będą dla nich
 {
  bladi2c();  //wyświetl komunikat o błędzie interfejsu I2C, jeśli nie było ACK
 }
 if(zapiszB_i2c(adrdane))    //zaadresuj rejestr w Slave'ie
 {
  bladi2c();  //wyświetl komunikat o błędzie interfejsu I2C, jeśli nie było ACK
 }
 for(;lz!=0;lz--)       //wysyłanie bloku danych (lz - liczba bajtów do wysłania
```

```
 {
  if(zapiszB_i2c(*pbufi2c++))     //wyślij kolejny bajt do Slave'a
  {
   bladi2c(); //wyświetl komunikat o błędzie interfejsu I2C, jeśli nie było ACK
  }
 }
 bitstopu();       //wyślij bit stopu (koniec transmisji)
}

void odi2c(unsigned char adri2c,unsigned char adrdane,unsigned char lz)
{       //odebranie bloku "lz" znaków od Slave'a I2C
        //"adri2c" - adres fizyczny Slave'a
        //"adrdane" - adres rejestru w Slave'ie, od którego będą odczytywane dane
 bitstartu();      //wyślij bit startu (początek transmisji)
 if(zapiszB_i2c(adri2c))     //wyślij adres Slave'a, to informacja dla wybranego
                        //Slave'a, że dane będą odbierane od niego
 {
  bladi2c();  //wyświetl komunikat o błędzie interfejsu I2C, jeśli nie było ACK
 }
 if(zapiszB_i2c(adrdane))    //zaadresuj rejestr w Slave'ie
 {
  bladi2c();  //wyświetl komunikat o błędzie interfejsu I2C, jeśli nie było ACK
 }
 bitstartu();      //ponowne wysłanie bitu startu
 if(zapiszB_i2c(adri2c+1))   //ponowne wysłanie adresu Slave'a z bitem R/W=1
                        //oznacza przełączenie Slave'a na nadawanie
 {
  bladi2c();  //wyświetl komunikat o błędzie interfejsu I2C, jeśli nie było ACK
 }
 pbufi2c=&bufi2c;  //ustaw wskaźnik bufora i2c na jego początek
 for(;lz>1;lz--)   //odbierz "lz" bajtów danych od Slave'a
 {
  *pbufi2c++=czytajB_i2c(1); //zapisuj odebrane bajty w buforze i2c
 }
 *pbufi2c++=czytajB_i2c(0);  //odbierz ostatni bajt od I2C, nie wysyłaj ACK
 bitstopu();       //wyślij bit stopu  (koniec transmisji)
}

SIGNAL(SIG_INTERRUPT0)       //procedura obsługi przerwania zewnętrznego INT0
{
 PORTB^=0x03;              //zmień stany LED1 i LED2
 odi2c(rtc,0x02,3);        //odczytaj czas z RTC (tylko godz, min i sek)
 fzegar=1;                 //ustaw flagę odczytania danych
}

void wyswietlczas(void)      //procedura wyświetlania czasu na LCD
{
 unsigned char zp;          //zmienna pomocnicza

 lcdxy(1,0);
 pbufi2c=&bufi2c[2];        //ustaw wskaźnik bufora na pozycje godzin
 zp=*pbufi2c--;             //pobierz godziny
 nalcd((zp&0x30)>>4,zp);    //wyświetl godziny
 piszznak(':');
 zp=*pbufi2c--;             //pobierz minuty
 nalcd((zp&0xf0)>>4,zp);    //wyświetl minuty
 piszznak(':');
 zp=*pbufi2c;               //pobierz sekundy
```

```
 nalcd((zp&0xf0)>>4,zp);     //wyświetl minuty
}

void zwolnijklaw(void)
{
 czekaj(200*tau);
 while((PIND&0x03)==0x03);   //czekaj aż wszystkie klawisze będą zwolnione
}

void ustawzegar(void)       //procedura ustawiania czasu w układzie RTC
{
 char *kom[1]={PSTR("STARTn/t")};

 cli();                 //wyłącz przerwania, aby nie naczytać danych
 piszilcd(0x0d);        //włącz mruganie
 odi2c(rtc,0x02,3);     //odczytaj czas z RTC (tylko godz, min, sek)
 wyswietlczas();
 pbufi2c=&bufi2c[1];    //ustaw wskaźnik bufora i2c na minuty
 bufi2c[0]=0;           //sekundy będą zawsze zerowane
 min=*pbufi2c++;        //pobierz minuty
 godz=*pbufi2c;         //pobierz godziny
 lcdxy(1,1);
 zwolnijklaw();
 while(sw4)       //czekaj na naciśnięcie SW4 wykonując poniższe instrukcje
 {              //ustawianie godzin
  lcdxy(1,1);
  if(!sw1)        //czy naciśnięto SW1?
  {
   godz++;        //inkrementuj godziny
   if((godz&0x0f)>9)
   {
    godz+=0x06;    //korekcja dziesiętna liczby BCD
   }
   if((godz&0x3f)>0x23)
   {
    godz&=0xc0;        //kasuj godziny po przekroczeniu zakresu
   }
   bufi2c[2]=(bufi2c[2]&0xc0)|godz;    //zapisz uaktualnione godziny w buforze i2c
                  //pozostawiając bity 24/12 i AM/PM nienaruszone
   wyswietlczas();
   czekaj(150*tau);
  }
 }
 lcdxy(1,4);
 zwolnijklaw();

 while(sw4)       //czekaj na naciśnięcie SW4 wykonując poniższe instrukcje
 {              //ustawianie minut
  lcdxy(1,4);
  if(!sw1)        //czy naciśnięto SW1?
  {
   min++;         //inkrementuj godziny
   if((min&0x0f)>9)
   {
    min+=0x06;     //korekcja dziesiętna liczby BCD
   }
   if(min>0x59)    //korekcja przekroczenia wartości 59
   {
```

```
    min&=0;   //przeniesienia na godziny nie ma, bo są one ustawiane niezależnie
   }
   bufi2c[1]=min;  //zapisz uaktualnione minuty w buforze i2c
   wyswietlczas();
   czekaj(150*tau);
  }
 }
 piszilcd(0x0c);        //wyłącz mruganie
 lcdxy(1,8);
 pisztekst(kom[0]);     //wyświetl komunikat "START"
 zwolnijklaw();
 while(sw1&sw4);        //czekaj na naciśnięcie jakiegoś klawisza
 if(!sw4)               //po naciśnięciu klawisza SW4...
 {
  doi2c(rtc,2,3);       //...zapisz ustawienia zegara
  bufi2c[0]=0;
  doi2c(rtc,0,1);       //start zegara
 }
 sei();           //włącz przerwania
}

int main(void)     //program główny
{
 unsigned char i,zp;

 //tablica komunikatów do wyświetlenia
 char *info[4]={
              PSTR("Ustaw zegar     "),
             PSTR("Blad I2C        "),
             PSTR("RTC  -  PCF8583 "),
             PSTR("ZL1AVR - plytka\newaluacyjna AVR")
             };
 DDRB=0xff;        //PORTB - wy
 PORTB|=0x02;      //zapal LED1, zgaś LED2
 DDRD=0x00;        //PORTD - we
 PORTD=0x07;       //podciąganie linii PD0, PD1 i PD2
 scl_wy=stan_0;    //SCL=0
 fzegar=0;

//>>>>>>>>>>>>>>>>>     Inicjowanie wyświetlacza LCD
<<<<<<<<<<<<<<<<<<<<<<<<<
 lcd_rs=0;
 czekaj(45*tau);        //opóźnienie ok. 45ms dla ustabilizowania się napięcia
                        //zasilania LCD (katalogowo min. 15 ms)
 for(i=0;i<3;i++)       //3-krotne wysłanie 3-
 {
  lcd_e=1;
  PORTB=(PORTB&0x0f)|0x30;   //wyślij 3- do LCD
  asm("nop");
  asm("nop");
  asm("nop");
  lcd_e=0;
  czekaj(5*tau);        //ok. 5ms
 }
 lcd_e=1;
 PORTB=(PORTB&0x0f)|0x20;    //wyślij 2- do LCD
 asm("nop");                 //wymagane wydłużenie impulsu
 asm("nop");
```

```
 asm("nop");
 lcd_e=0;              //impuls strobujący
 czekaj(10L);
 piszilcd(0x28);       //interfejs 4-bitowy, 2 linie, znak 5x7
 piszilcd(0x08);       //wyłącz LCD, wyłącz kursor, wyłącz mruganie
 piszilcd(0x01);       //czyść LCD
 czekaj(1.64*tau);     //wymagane dla instrukcji czyszczenia ekranu opóźnienie
 piszilcd(0x06);       //bez przesuwania w prawo
 piszilcd(0x0c);       //włącz LCD, bez kursora, bez mrugania
//>>>>>>>>>>>>>>>>>>>           <<<<<<<<<<<<<<<<<<<<<<<<<<<

 GIMSK=1<<INT0;        //włącz przerwanie zewnętrzne INT0
 MCUCR=0x02;           //przerwania INT0 wyzwalane opadającym zboczem
 if(!sw1)              //czy w chwili restartu był naciśnięty SW1?
 {
  czysclcd();
  pisztekst(info[3]);  //wyświetl info o płytce
  czekaj(1000*tau);
  while(sw1);          //czekaj na ponowne naciśnięcie SW1
  czysclcd();
 }
 pisztekst(info[2]);   //wyświetl komunikat o układzie RTC na LCD
 sei();                //włącz przerwania
 while(1)
 {
  if(fzegar)           //czy można odświeżyć czas na LCD
  {
   fzegar=0;
   lcdxy(1,0);
//   odi2c(rtc,0x02,3);
   wyswietlczas();
  }
  if(!sw4)             //czy naciśnięto klawisz SW4
  {
   czekaj(150*tau);
   lcdxy(0,0);
   pisztekst(info[0]); //wyświetl komunikat o trybie ustawiania
   ustawzegar();       //ustaw zegar
   czysclcd();
   pisztekst(info[2]); //odtwórz komunikat o układzie RTC na LCD
   czekaj(300*tau);
  }
 }
}
```

```
/*********************************************************************/
/* Ćwiczenie 11 - Połączenie mikrokontrolera AVR do komputera PC    */
/*              poprzez port USB                                     */
/*              Nadawanie i odbiór poprzez UART z użyciem przerwań*/
/* J.D. '2003                                                       */
/*********************************************************************/

#include <io.h>
#include <progmem.h>
#include <stdlib.h>
#include <interrupt.h>
#include <signal.h>

#define FCPU      8000000    //częstotliwość oscylatora CPU
#define VUART     38400      //prędkość transmisji [b/s]
#define VUBRR     FCPU/(VUART*16)-1    //wpis do UBRR dla VUART

//  Poniższe definicje służą do realizacji wygodnego dostępu bitowego
typedef struct _bit_struct
{
 unsigned char bit0: 1;
 unsigned char bit1: 1;
 unsigned char bit2: 1;
 unsigned char bit3: 1;
 unsigned char bit4: 1;
 unsigned char bit5: 1;
 unsigned char bit6: 1;
 unsigned char bit7: 1;
}pole_bitowe;

#define DAJ_BIT(adr) (*((volatile pole_bitowe*) (adr)))
#define _PORTB 0x38
#define _PINB 0x36
#define _PORTD 0x32
#define _DDRD 0x31
#define _PIND 0x30
#define lcd_rs DAJ_BIT(_PORTB).bit2
#define lcd_e DAJ_BIT(_PORTB).bit3
#define led0 DAJ_BIT(_PORTB).bit0
#define led1 DAJ_BIT(_PORTB).bit1
#define LF 0x0a    //definicja znaku LF (przecie do nowej linii)
#define CR 0x0d    //definicja znaku CR (powrót karetki)

unsigned char wiersz=0;      //pozycja umieszczenia znaku na LCD
unsigned char kolumna=0;     //pozycja umieszczenia znaku na LCD

char volatile iofifosio;     //wskaźnik odczytu kolejki UART-u
char volatile izfifosio;     //wskaźnik zapisu kolejki UART-u
char fifosio[32];            //kolejka UART-u
unsigned char volatile ldanych=0;      //liczba danych w buforze fifosio

void czekaj(unsigned long zt)     //procedura wytracania czasu
{
 #define tau 10.38           //przybliżony przelicznik argumentu na ms
 unsigned char zt1;
 for(;zt>0;zt--)
 {
  for(zt1=255;zt1!=0;zt1--);
```

```
 }
}

//>>>>>>>>>>>>>   Procedury obsługi wyświetlacza LCD   <<<<<<<<<<<<<<<<<<
void piszilcd(unsigned char instr)     //zapisz instrukcję sterującą do LCD
{
 lcd_rs=0;
 lcd_e=1;
 PORTB=(PORTB&0x0f)|(instr&0xf0);  //przygotuj starszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD ok. 100us
 lcd_e=1;
 PORTB=(PORTB&0x0f)|((instr&0x0f)<<4); //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD ok. 100us
}

void piszdlcd(char dana)     //zapisz daną do LCD
{
 lcd_rs=1;
 lcd_e=1;
 PORTB=(PORTB&0x0f)|(dana&0xf0);  //przygotuj starszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD
 lcd_e=1;
 PORTB=(PORTB&0x0f)|((dana&0x0f)<<4);  //przygotuj młodszy półbajt do LCD
 asm("nop");
 asm("nop");
 asm("nop");
 lcd_e=0;          //impuls strobujący
 czekaj(10L);      //czekaj na gotowość LCD
}

void czysclcd(void)          //czyść ekran
{
 piszilcd(0x01);   |//polecenie czyszczenia ekranu dla kontrolera LCD
 czekaj(1.64*tau); //rozkaz 0x01 wykonuje się 1.64ms
 wiersz=0;
 kolumna=0;
}

void lcdxy(unsigned char w, unsigned char k)     //ustaw współrzędne kursora
{
 piszilcd((w*0x40+k)|0x80);  //standardowy rozkaz sterownika LCD
}                            //ustawiający kursor w określonych współrzędnych

void piszznak(char znak)     //procedura umieszcza znak na wyświetlaczu
{
 piszdlcd(znak);             //wyświetl znak na LCD
```

```
}

void pisztekst(char *tekst,unsigned char romram) //pisz tekst na LCD wskazywany
                                                 //pointerem *tekst
{
 char zn;
 char nr=0;

 while(1)
 {
  if(!romram)
  {
   zn=PRG_RDB(&tekst[nr++]);      //pobranie znaku z pamięci programu
  }
  else
  {
   if(!ldanych)              //czy są jeszcze jakieś dane w buforze
   {
    break;
   }
   zn=fifosio[iofifosio];    //pobierz znak z bufora
   iofifosio==31?iofifosio=0:++iofifosio;   //inkrementacja modulo 32
   ldanych--;
  }
  if(zn!=0)                  //czy nie ma końca tekstu?
  {
   if((zn==LF)||(zn==CR))    //czy znak nowej linii?
   {

    if(zn==LF)
    {
     wiersz==1?wiersz=0:++wiersz;      //przejdź do nowej linii
    }
    else          //był CR
    {
     kolumna=0;    //powrót karetki
    }
   }
   else
   {
    piszdlcd(zn);            //umieć pojedynczy znak tekstu na LCD
    kolumna==15?kolumna=0:++kolumna;   //inkrementacja modulo 16
    if(kolumna==0)
    {
     wiersz==1?wiersz=0:++wiersz;      //przejdź do nowej linii
    }
   }
   lcdxy(wiersz,kolumna);
  }
  else
  {
   break;                    //zakończ pętlę, jeśli koniec tekstu
  }
 }
}

SIGNAL(SIG_UART_RECV)        //procedura obsługi odbiornika UART-u
{
```

```
 led1=0;
 if(ldanych<32)
 {
  fifosio[izfifosio]=UDR;                 //zapamiętaj odebrany znak
  izfifosio==31?izfifosio=0:++izfifosio;   //inkrementacja modulo 32
  ldanych++;
 }
 led1=1;                  //zgaś leda
}

SIGNAL(SIG_UART_TRANS)      //procedura obsługi nadajnika UART
{                          //wywoływana po wysłaniu znaku
 char znak;

 led0=0;
 znak=fifosio[iofifosio];   //pobierz dane z pamięci RAM
 if(ldanych!=0)             //czy koniec pobierania danych?
 {
  UDR=znak;                 //nie, wyślij znak pobrany z kolejki
  ldanych--;
 }
 else
 {
  cbi(UCR,TXEN);            //tak, wyłącz nadajnik
 }
 iofifosio==31?iofifosio=0:++iofifosio;    //inkrementacja mod 32
 led0=1;
}

int main(void)
{
 unsigned long i;
 char znak;

 //tablica komunikatów do wysłania
 char *info[2]={
            PSTR("Wysylam dane"),
             PSTR("Przeslij z PC-ta")
            };

 DDRB=0xff;   //PORTB - wy
 PORTB=0xff;
 DDRD=0x02;   //PD1 - wy (RXD), pozostałe we
 PORTD=0x02;  //podciągania wejścia PD1 (RXD)

 lcd_rs=0;
 czekaj(45*tau);       //opóźnienie ok. 45ms dla ustabilizowania się napięcia
                      //zasilania LCD (katalogowo min. 15 ms)
 for(i=0;i<3;i++)      //3-krotne wysłanie 3-
 {
  lcd_e=1;
  PORTB=(PORTB&0x0f)|0x30;  //wyślij 3- do LCD
  asm("nop");
  asm("nop");
  asm("nop");
  lcd_e=0;
  czekaj(5*tau);       //ok. 5ms
 }
```

```
lcd_e=1;
PORTB=(PORTB&0x0f)|0x20;    //wyślij 2- do LCD
asm("nop");                 //wymagane wydłużenie impulsu
asm("nop");
asm("nop");
lcd_e=0;              //impuls strobujący
czekaj(10L);
piszilcd(0x28);       //interfejs 4-bitowy, 2 linie, znak 5x7
piszilcd(0x08);       //wyłącz LCD, wyłącz kursor, wyłącz mruganie
piszilcd(0x01);       //czyść LCD
czekaj(1.64*tau);     //wymagane dla instrukcji czyszczenia ekranu opóźnienie
piszilcd(0x06);       //bez przesuwania w prawo
piszilcd(0x0c);       //włącz LCD, bez kursora, bez mrugania

pisztekst(info[0],0);  //komunikat na LCD
UBRR=VUBRR;            //ustaw prędkość transmisji
UCR=1<<RXCIE | 1<<TXCIE  | 1<<RXEN;   //zezwolenie na przerwania od
            //odbiornika i nadajnika, zezwolenie na odbiór i nadawanie
sei();          //włącz przerwania

izfifosio=0;      //inicjuj zmienne
iofifosio=0;
znak=' ';
sbi(UCR,TXEN);         //włącz nadajnik
UDR=znak++;            //umieść pierwszy znak
for(i=0;i<0x1ffff;i++) //pętla wysyłania znaków
{
 while(ldanych==32);   //jeśli bufor przepełniony, to czekaj
 fifosio[izfifosio]=znak++;      //umieść znak do wysłania w buforze
 if(znak>'Z')
 {
  znak=' ';
 }
 izfifosio==31?izfifosio=0:++izfifosio;    //inkrementuj mod 32
 ldanych++;
}
czysclcd();
pisztekst(info[1],0);
czekaj(2000*tau);
czysclcd();

izfifosio=0;      //inicjuj zmienne
iofifosio=0;
ldanych=0;
while(1)          //pętla odbioru znaków
{
 if(ldanych)      //czy odebrano jakiś znak?
 {
  pisztekst(fifosio[0],1);  //wyświetl odebrane znaki
 }
}
}
```

Kilka różnych porad

Przy rozpoczynaniu działalności z **avr-gcc** i mikrokontrolerami **Atmega** możemy natknąć się na kilka niespodzianek, które potrafią pochłonąć sporo czasu i nerwów - więc lepiej wiedzieć o nich wcześniej.

- uwaga na bity konfiguracji (*fuses*) :

Mikrokontrolery *Atmega* dostarczane są w konfiguracji pracy z wewnętrznym oscylatorem 1MHz. Może to prowadzić do kłopotów z timerami, uartem itp. o ile zapomnimy przestawić fuse'y dla uzyskania częstotliwości pracy potrzebnej w projekcie (mogą także wymagać odpowiedniej konfiguracji niektóre programatory).

Jeśli używamy wewnętrznego oscylatora pamiętajmy o rejestrze kalibracyjnym *OSCCAL*. Atmega ładuje go sprzętowo po resecie zawsze wartością dla 1 MHz - niezależnie od ustawionego fuse'ami taktowania. Dla innych częstotliwości program musi samodzielnie przeładować *OSCCAL* odpowiednią wartością. Nie jest ona dostępna w trakcie normalnej pracy a tylko w trybie programowania - musi być więc wcześniej odczytana z kostki i zapisana. Wygodnie jest wyposażyć programator w automatyzację tego procesu (np. współpracujący z *AvrSide* programator usb samoczynnie przepisuje odpowiedni bajt kalibracyjny do ostatniej komórki pamięci eeprom, skąd później program odczytuje wartość instrukcją :

```
OSCCAL=eeprom_read_byte((uchar*)E2END);
```

W niektórych kostkach możliwe jest użycie pinu *RESET* jako linii I/O. Jednak ustawienie tej opcji uniemożliwi dalsze programowanie szeregowe - mikrokontroler wymaga wtedy zastosowania tzw. programatora wysokonapięciowego pozwalającego na bardziej wszechstronne operacje.

Kłopotliwe może być też niewłaściwe ustawienie źródła taktowania (np. zewnętrznym zegarem) - jest to wprawdzie odwracalne ale wymaga dodatkowych zabiegów, niekiedy (np. przy gotowej płytce smd) dosyć uciążliwych.

Atmega128 jest dostarczona z zaprogramowanym trybem kompatybilności z modelem 103, z czym wiąże się mniejszy obszar RAM oraz niedostępność rozszerzonych rejestrów. *Avr-gcc* samoczynnie inicjalizuje stos na końcu RAM - a więc w trybie kompatybilności poza rzeczywiście dostępnym sprzętowo obszarem. Objawia się to trudnym do zinterpretowania wywracaniem się programu przy pierwszym przerwaniu czy wywołaniu funkcji, chociaż pierwsze testy wykazały poprawność pracy układu.

Największe Atmegi (128 i 64) mają - w odróżnieniu do całej pozostałej rodziny - wydzielone piny (MOSI i MISO) dla programowania szeregowego. Jest to bardzo korzystne w przypadku użycia na płycie układów SPI mogących zakłócać przebieg programowania, ale może też prowadzić do wynikającej z przyzwyczajeń pomyłki.

W Atmegach dysponujących interfejsem *JTAG* jest on domyślnie włączony. Powoduje to zablokowanie zwykłych funkcji I/O odpowiednich pinów - jeśli chcemy z nich korzystać musimy wyłączyć (ustawić 1) *fuse* JTAGEN.

- uwaga na przerwania:

Avr-gcc posiada dwa typy handlerów przerwań : ***SIGNAL()*** - utrzymujący sprzętowe wyłączenie przerwania (wszelkie pozostałe przerwania są zablokowane do momentu zakończenia obsługi bieżącego); oraz ***INTERRUPT()*** - wyposażony w prologu w odblokowanie się przerwań. Zasadniczym błędem jest użycie ***INTERRUPT*** w przypadku przerwań wyzwalanych warunkiem. Klasyczny przykład to np. obsługa usart - przerwanie jest aktywne zawsze po odebraniu bajtu, aż do jego odczytania. Jednak ***sei()*** w prologu nie daje nam szansy odczytu rejestru odbiornika - wcześniej zawsze wystąpi ponowne wywołanie handlera - i tak kolejno aż do przepełnienia stosu i wywrócenia się programu.

Musimy dokładnie wpisywać nazwę przerwania. ***Avr-gcc*** nie sprawdza poprawności - błąd spowoduje brak odpowiedniego skoku w tablicy wektorów przerwań bez żadnego ostrzeżenia. W razie wątpliwości należy zajrzeć do kodu asm i pliku nagłówka. AvrSide znacznie ułatwia sprawę - w szablonach funkcji ***avr-libc*** jest pełny wykaz przerwań, poza tym dla poprawnej nazwy wyświetlana jest podpowiedź a dla błędnej nie.

Przy włączonej optymalizacji zmienne globalne, które chcemy zmieniać w obsłudze przerwań muszą koniecznie posiadać atrybut ***volatile***. W przeciwnym razie kompilator zignoruje zmiany.

- uwaga na dostęp do pamięci:

Dostęp do pamięci **eeprom** oraz **flash** (np. tablice czy teksty) jest realizowany inaczej niż w dedykowanych kompilatorach (uniwersalny, wieloplatformowy gcc nie do końca daje się przystosować do architektury AVR z rozdzielonymi obszarami pamięci). Zapis i odczyt z eeprom oraz odczyt z flash wymagają - zamiast bezpośrednich przypisań - używania dodatkowych makr i atrybutów. Np. tradycyjny zapis :

```
const char X = 2;
char a;
```

i późniejsza próba odczytu tej wartości:

```
a = X;
```

nie dadzą prawidłowego wyniku. Należy użyć zapisu:

```
char X PROGMEM = 2;
```

i później odczytywać wartość z flasha makrem:

```
a = pgm_read_byte(&X);
```

- uwagi różne

Avr-gcc nie posiada składni bezpośredniego dostępu do bitów *zmienna.numer_bitu*. Operacje na bitach wykonujemy używając iloczynu i sumy bitowej, np. dla bajtu:

```
zmienna |= _BV(numer_bitu); // ustawienie bitu
zmienna &= ~_BV(numer_bitu); //zgaszenie bitu
zmienna ^= _BV(numer_bitu); // przełączenie bitu
```

Standard C nie przewiduje formatu binarnego stałych - więc np. ***00110011*** zapisujemy albo od razu w formacie hex (***0x33***) albo używając makra *_BV*. Jest to czasem kłopotliwe, dlatego obecnie udostępnione jest rozszerzenie avr-gcc wprowadzające możliwość bezpośredniego zapisu stałych binarnie (***0b00110011***).

Mikrokontrolery **AVR** nie posiadają znanego z rodziny **'51** obszaru pamięci adresowanego bitowo i związanego z tym naturalnego wsparcia dla jednobitowych, oszczędzających pamięć zmiennych logicznych. Możemy jednak uzyskać ten sam efekt wykorzystując standardowe mechanizmy C i deklarując unię, która zawiera albo bajt albo strukturę ośmiu pól jednobitowych :

```
typedef struct
{
uchar Flag1:1;
uchar Flag2:1;
uchar Flag3:1;
uchar Flag4:1;
uchar Flag5:1;
uchar Flag6:1;
uchar Flag7:1;
uchar Flag8:1;
} FlagBits;

typedef union
{
FlagBits Bits;
uchar   Byte;
} Flags;
```

Do zmiennych typu *Flags* mamy wtedy dostęp albo poprzez cały bajt albo poprzez poszczególne bity. Wygodnie jest też zdefiniować bity jako flagi nazwane zgodnie z pełnioną funkcją - ułatwia to ich używanie w kodzie :

```
volatile Flags SysFlags;
volatile Flags SerialFlags;
#define MS100FLAG SysFlags.Bits.Flag1
#define T1_DELAY_DONE SysFlags.Bits.Flag2
```

Używając standardowego pliku nagłówkowego *stdbool.h* możemy też posługiwać się oznaczeniami *true* i *false* (po prostu 1 i 0 ale dużo bardziej czytelne w kodzie) i pisać np.

```
MS100FLAG = true;
```

Z tych rozbudowanych zapisów avr-gcc tworzy bardzo zwarty i oszczędny kod. Jednak jest i wada: format *coff* nie przenosi informacji o polach bitowych - nie możemy więc bezpośrednio podglądać tak zdefiniowanych flag w debugerze *AvrStudio*.

Przy dołączaniu funkcji bibliotecznej linker przeszukuje archiwum *.a, znajduje plik *.o z kodem żądanej funkcji i cały ten plik włącza do kodu niezależnie od aktualnej przydatności w programie jego pozostałej treści. Może to dość znacznie i zbytecznie zwiększyć zużycie pamięci. Niedogodność tę zazwyczaj eliminuje się przy tworzeniu własnych bibliotek poprzez umieszczanie pojedynczych funkcji w oddzielnych modułach i kompilowanie ich do oddzielnych plików *.o (tak są zbudowane systemowe biblioteki *avr-gcc* co możemy łatwo obejrzeć w managerze bibliotek *AvrSide*).

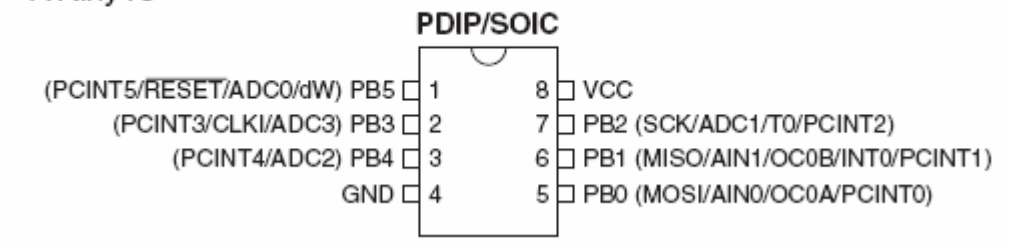
Dodatki

Zawarte tutaj informacje mają pozwalać na szybkie i łatwe znalezienie potrzebnych informacji praktycznych.

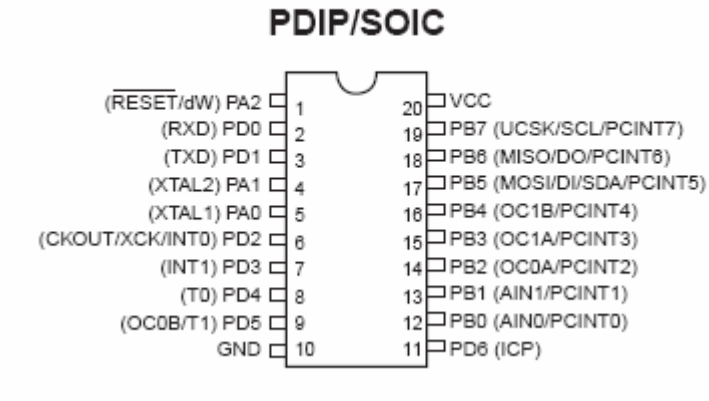
Opisy wyprowadzeń mikrokontrolerów Atmela AVR

Poniżej przedstawiono opis wyprowadzeń najpopularniejszych mikrokontrolerów AVR

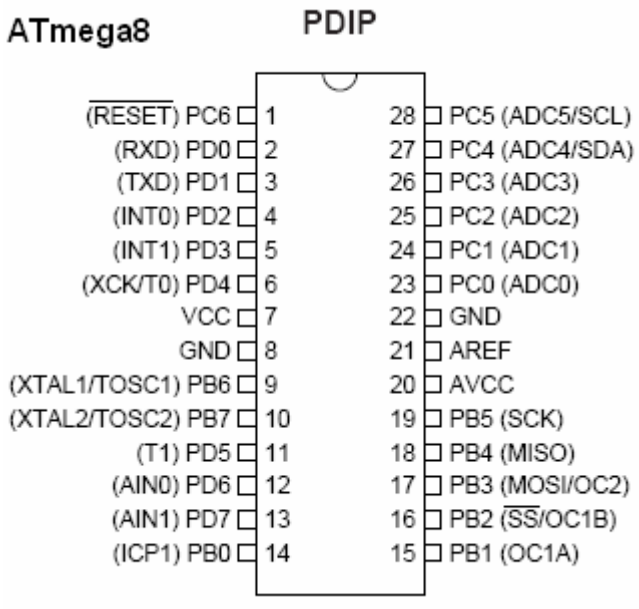
ATtiny13

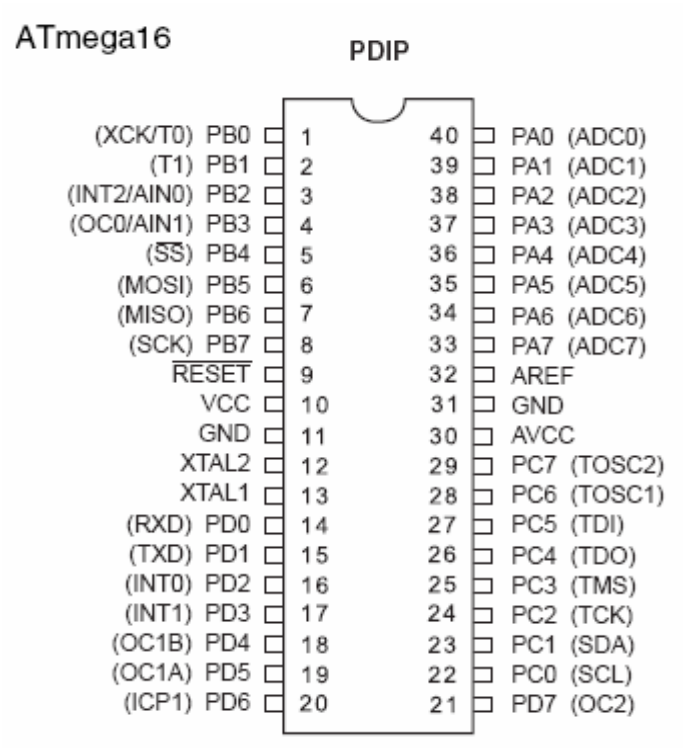


ATtiny2313



ATmega8





Konfiguracja fusebitów w uC AVR

Fuse i *lock* bity niosą wiadomości najważniejsze dla procesora. Między innymi z jaką częstotliwością ma pracować, czy jest to częstotliwość z wewnętrznego generatora czy zewnętrznego, jak dany procesor ma być programowany itp (to *fuse* bity). *Lock* bity odpowiedzialne są za blokowanie odczytu lub zapisu pamięci flash, eeprom, np. wtedy gdy chcemy aby program z procesora nie mógłbyć skopiowany i wykorzystany przez kogoś niepowołanego. Poniżej znajduje się przykład przedstawiający konfiguracje fuse bitów.

```
# Fuse high byte:
# 0xc9 = 1 1 0 0 1 0 0 1 <-BOOTRST (boot reset vector at 0x0000)
#        ^ ^ ^ ^ ^ ^ ^ ^--- BOOTSZ0
#        | | | | | +--------   BOOTSZ1
#        | | | | + --------   EESAVE (don't preserve EEPROM over chip erase)
#        | | | +-------------- CKOPT (full output swing)
#        | | +---------------- SPIEN (allow serial programming)
#        | +------------------ WDTON (WDT not always on)
#        +-------------------- RSTDISBL (reset pin is enabled)
# Fuse low byte:
# 0x9f = 1 0 0 1 1 1 1 1
#        ^ ^ \ / \--+--/
#        | | |     +-------  CKSEL 3..0 (external >8M crystal)
#        | | +---------------- SUT 1..0 (crystal osc, BOD enabled)
#        | +------------------ BODEN (BrownOut Detector enabled)
#        +-------------------- BODLEVEL (2.7V)
```

Jedynka oznacza opcję nieużywaną, zaś 0 opcję zaznaczoną. *High byte* tyczą się opcji procesora takich jak: rozmiar buforowania, rodzaj programowania, użycie pinu resetu jako standardowego IO. *Low byte* to przede wszystkim źródło i rodzaj taktowania, oraz opcje oszczędzania energii.

Pliki konfiguracyjne AVRdude

Plikiem konfiguracyjnym programu *AVRdude* jest plik *avrdude.conf* znajdujący się w katalogu z tymże programem. W pliku tym zawarte są wiadomości o wszystkich opcjach, identyfikatorach wykorzystywanych w programie. Poniżej przedstawiono podstawowe identyfikatory dla procesorów i programatorów. Oczywiście nowsze wersje *AVRdude* mogą różnić się ilością zarówno dostępnych mikrokontrolerów jak i programatorów.

Spis identyfikatorów dla uC

c128	AT90CAN128
pwm2	AT90PWM2
pwm3	AT90PWM3
1200	AT90S1200
2313	AT90S2313
2333	AT90S2333
2343	AT90S2343
4414	AT90S4414
4433	AT90S4433
4434	AT90S4434
8515	AT90S8515
8535	AT90S8535
m103	ATmega103
m128	ATmega128
m1280	ATmega1280
m1281	ATmega1281
m16	ATmega16
m161	ATmega161
m162	ATmega162
m163	ATmega163
m164	ATmega164
m169	ATmega169
m2560	ATmega2560
m2561	ATmega2561
m32	ATmega32
m324	ATmega324
m329	ATmega329
m3290	ATmega3290
m48	ATmega48
m64	ATmega64
m640	ATmega640
m644	ATmega644
m649	ATmega649
m6490	ATmega6490
m8	ATmega8
m8515	ATmega8515
m8535	ATmega8535
m88	ATmega88
t12	ATtiny12
t13	ATtiny13
t15	ATtiny15
t2313	ATtiny2313
t25	ATtiny25
t26	ATtiny26
t45	ATtiny45
t85	ATtiny85

Spis identyfikatorów dla programatorów

abcmini	ABCmini Board,
avr109	Atmel AppNote AVR109 Boot Loader
avr910	Atmel Low Cost Serial Programmer
avr911	Atmel AppNote AVR911 AVROSP
avrisp	Atmel AVR ISP (an alias for stk500)
avrispv2	Atmel AVR ISP
avrispmkII	Atmel AVR ISP mkII
avrispmk2	Atmel AVR ISP mkII
bascom	Bascom SAMPLE programming cable
bsd	Brian Dean's Programmer
butterfly	Atmel Butterfly Development Board
dt006	Dontronics DT006
dragon_isp	AVR Dragon in ISP mode
dragon_jtag	AVR Dragon in JTAG mode
frank-stk200	Frank's STK200 clone,
jtagmkI	Atmel JTAG ICE
jtag1slow	Atmel JTAG ICE mkI
jtagmkII	Atmel JTAG ICE mkII
jtag2slow	Same as before.
jtag2fast	Atmel JTAG ICE mkII,
jtag2	Same as before.
jtag2isp	Atmel JTAG ICE mkII in ISP mode.
jtag2dw	Atmel JTAG ICE mkII in debugWire mode.
pavr	Jason Kyle's pAVR Serial Programmer
pony-stk200	Pony Prog STK200
sp12	Steve Bolt's Programmer
stk200	STK200
stk500	Atmel STK500
stk500v1	Atmel STK500
stk500hvsp	Atmel STK500 in high-voltage serial
stk500pp	Atmel STK500 in parallel programming mode
stk500v2	Atmel STK500

Bibliografia

1. **Spis plików pdf**: *avrdude.pdf*, *atmega8.pdf*, *atmega16.pdf*, *attiny2323.pdf*, *attiny13.pdf* i inne
2. **Listingi** pochodzą z książki *"Mikrokrokontrolery AVR w praktyce"* J.D.
3. **Opis ćwiczeń** i biblioteki avr-libc ze strony internetowej *avr.elportal.pl*
4. **Pozostałe** wiadomości pochodzą z dokumentacji technicznych dostarczanych wraz z pakietem *WinAVR*, dokumentacji mikroprocesorów *Atmel*.