

Podstawy systemów mikroprocesorowych

Wykład nr 2

dr Piotr Fronczak

<http://www.if.pw.edu.pl/~agatka/psm.html>

fronczak@if.pw.edu.pl

Pokój 6GF

Dotychczas program wyglądał mniej więcej tak:

```
#include <stdio.h>
int main()
{
    printf("Hello World!");
    return 0;
}
```

Odtąd powinien wyglądać tak:

```
void main()
{
    while (1); // do końca świata...
}
```

Typowy program

```
#include <avr/io.h>
#include <util/delay.h>
#include „lcd.h”
#define CZAS 100
int x;
uint8_t b;

int main(void)
{
    DDRD=0xFF;
    for(;;) {                // wieczna pętla (albo tak: while(1){})
        PORTD^=0xFF;        // migamy
        _delay_ms(CZAS);    // czekamy 100 milisekund
    }
    return 0;                // kod powrotu
}
```

Pliki biblioteczne dołączamy za pomocą dyrektywy **#include**.

```
#include <avr/io.h>
#include <avr/delay.h>
#include "lcd.h"          // nasze własne funkcje
```

Pliki nagłówkowe zawierają deklaracje zmiennych i funkcji jak również polecenia z wykorzystaniem dyrektywy **#DEFINE**.

Same definicje funkcji są w bliźniaczym pliku *.c

plik lcd.h

```
#ifndef LCD_H_
#define LCD_H_

void lcd_init(void);

#endif
```

plik lcd.c

```
#include <avr/io.h>
#include "lcd.h"

void set_lcd(void)
{
    // coś tam
}

void lcd_init(void)
{
    set_lcd();
}
```

Typowy program

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

#define CZAS 100

int x;
uint8_t b;

int main(void)
{
    DDRD=0xFF;
    for(;;) {                // wieczna pętla (albo tak: while(1){})
        PORTD^=0xFF;        // migamy
        _delay_ms(CZAS);    // czekamy 100 milisekund
    }
    return 0;                // kod powrotu
}
```

Definicje

Dyrektywa **#DEFINE**.

Makra preprocesora. Ich argumenty podstawiane do kodu przed kompilacją. Dobry nawyk – drukowana czcionka.

```
#define F_CPU 16000000    // szybkość CPU w hercach
#define TRUE  0x01        // logiczne TRUE
#define FALSE 0x00        // logiczne FALSE
```

```
#define LICZ(a,b) a+b
```

```
a = 2 * LICZ(5,2);      ----->      a = 2 * 5 + 2;
```

```
#define LED 0b00010000
PORTC = LED;
```

Typowy program

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

```
#DEFINE CZAS 100
```

```
int x;
uint8_t b;
```

Deklarujemy zmienne globalne

```
int main(void)
{
    DDRD=0xFF;
    for(;;) {                // wieczna pętla (albo tak: while(1){})
        PORTD^=0xFF;        // migamy
        _delay_ms(CZAS);    // czekamy 100 milisekund
    }
    return 0;                // kod powrotu
}
```

Zmienne i typy

Programując mikrokontrolery należy zwracać uwagę na różnice w definicjach typów zmiennych (**int** może znaczyć różne rzeczy).

Warto korzystać z typów zdefiniowanych w pliku nagłówkowym **stdint.h**.

Kilka przykładów:

```
typedef signed char      int8_t;  
typedef unsigned char   uint8_t;  
typedef short           int16_t;  
typedef unsigned short  uint16_t;  
typedef long            int32_t;  
typedef unsigned long   uint32_t;  
typedef long long       int64_t;  
typedef unsigned long long uint64_t;
```

Unikamy typów zmiennoprzecinkowych (bo są pamięciożerne).
Brak typu **double**.

Zmienne i typy

- Kwantem pamięci mikroprocesora jest bajt. Zatem najmniejszy rozmiar odczytanej (lub zapisanej danej) to 1 bajt (8-bitów).
- Typ bool również zajmuje 1 bajt.
- Gdy mamy dużo zmiennych typu bool warto zebrać je w strukturę.

Struktury w C mogą zawierać także pola zajmujące mniej niż 1 bajt. Aby zadeklarować takie pole, należy podać po dwukropku liczbę bitów.

```
struct liczba {  
    unsigned int flaga_1 :1;  
    unsigned int flaga_2 :1;  
    unsigned int flaga_dwubitowa :2;  
} x;
```

Bity zapisujemy tak:

```
x.flaga_1 = 1;  
x.flaga_2 = 0;  
x.flaga_dwubitowa = 3;
```

Typowy program

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

```
#DEFINE CZAS 100
```

```
int x;
uint8_t b;
```

```
int main(void)
```

```
{
```

```
    DDRA=0xFF;
```

```
    for(;;) {           // wieczna pętla (albo tak: while(1){})
```

```
        PORTA^=0xFF;    // migamy
```

```
        _delay_ms(CZAS); // czekamy 100 milisekund
```

```
    }
```

```
    return 0;           // kod powrotu
```

```
}
```

zawsze

Typowy program

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

```
#DEFINE CZAS 100
```

```
int x;
uint8_t b;
```

```
int main(void)
```

```
{
```

```
    DDRA=0xFF;
```

```
    for(;;) {
```

```
        PORTA^=0xFF;
```

```
        _delay_ms(CZAS);
```

```
    }
```

```
    return 0;
```

```
}
```

Ustawiamy rejestry

```
// wieczna pętla (albo tak: while(1){})
```

```
// migamy
```

```
// czekamy 100 milisekund
```

```
// kod powrotu
```

io.h

```
...
#elif defined (__AVR_ATmega3290__)
# include <avr/iom3290.h>
#elif defined (__AVR_ATmega3290P__)
# include <avr/iom3290.h>
#elif defined (__AVR_ATmega32HVB__)
# include <avr/iom32hvb.h>
#elif defined (__AVR_ATmega406__)
# include <avr/iom406.h>
#elif defined (__AVR_ATmega16__)
# include <avr/iom16.h>
#elif defined (__AVR_ATmega16A__)
# include <avr/iom16a.h>
#elif defined (__AVR_ATmega161__)
# include <avr/iom161.h>
#elif defined (__AVR_ATmega162__)
# include <avr/iom162.h>
#elif defined (__AVR_ATmega163__)
# include <avr/iom163.h>
#elif defined (__AVR_ATmega164P__)
...

```

iom16.h

```
/* PORTA */
#define PORTA      _SFR_IO8(0x1B)
#define PA7        7
#define PA6        6
...
#define PA0        0

/* DDRA */
#define DDRA       _SFR_IO8(0x1A)
#define DDA7       7
#define DDA6       6
...
#define DDA0       0

/* PINA */
#define PINA       _SFR_IO8(0x19)
#define PINA7      7
#define PINA6      6
...
#define PINA0      0

/* UCSR1A */
#define RXC1       7
#define TXC1       6
#define UDRE1      5
#define FE1        4
#define DOR1       3
#define UPE1       2
#define U2X1       1
#define MPCM1      0

```

Port A Data Register – PORTA

Bit	7	6	5	4	3	2	1	0	
	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Port A Data Direction Register – DDRA

Bit	7	6	5	4	3	2	1	0	
	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Port A Input Pins Address – PINA

Bit	7	6	5	4	3	2	1	0	
	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	PINA
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

```
// ustawiamy wszystkie piny portu A jako wyjściowe  
DDRA = 0xFF;  
DDRA = 0b11111111;
```

```
// inny przykład: Sterujemy diodą za pomocą pinu 5  
portu A, a badamy stan przycisku za pomocą pinu 7  
  
DDRA = 0b0x1xxxxx;
```

Za miesiąc pin nr 5 przyda nam się do czegoś innego. Jak poprawić kod?

Operacje bitowe

Operacje bitowe ułatwiają modyfikacje pojedynczych bitów w słowie dzięki maskowaniu i przesunięciom.

- Zaprzeczenie NOT: \sim
- Iloczyn AND: $\&$
- Suma OR: $|$
- Suma wyłączająca XOR: \wedge
- Przesunięcie w prawo: $>>$
- Przesunięcie w lewo: $<<$

Operator AND

char x = 101¹ 1100; Chcemy zmienić bit nr 4 na 0.

char y = 1110 1111; Wykorzystamy y jako maskę.

char z;

z = x & y;

$$\begin{array}{r} 1011\ 1100 \\ \text{AND } 1110\ 1111 \\ \hline 101\mathbf{0}\ 1100 \end{array}$$

Maska i operator & zmieniają stan bitu na 0

Operator OR

char x = 1000 0101; Chcemy ustawić bit nr 4 na 1.

char y = 0001 0000; Używamy y jako maski

char z;

z = x | y;

$$\begin{array}{r} 1000\textcolor{red}{0} 0101 \\ \text{OR } 0001 0000 \\ \hline 100\textcolor{red}{1} 0101 \end{array}$$

Maska i operator $|$ zmieniają stan bitu na 1.

Operator SHIFT

```
char x = 0000 0001;
```

```
char y = 5;
```

```
char z;
```

```
z = x << 5;
```

$$\begin{array}{r} 00000001 \\ \hline 00100000 \end{array}$$

Operator << przesuwa x o y miejsc w lewo.

Operator >> przesuwa x o y miejsc w prawo.

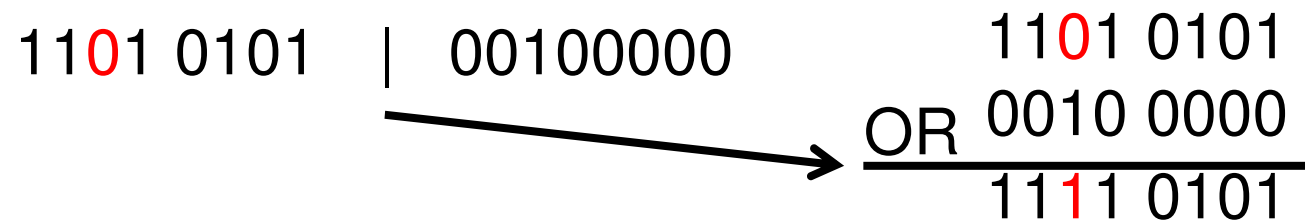
Operator OR i SHIFT

`char wynik = 1101 0101;` Chcemy zmienić 5 bit na 1

`char maska = 0000 0001;`

`char przesuniecie = 5;`

`wynik = wynik | (maska << przesuniecie);`



Przesunięcie jedynki (albo jedynek) maski na właściwą pozycję i wykonanie operacji **OR** ustawia odpowiednie bity na 1.

`wynik |= (maska << przesuniecie);`

Operator AND i SHIFT

char wynik = 1011 0101; Chcemy zmienić bit nr 5 na 0.

char maska = 0000 0001;

char przesuniecie = 5;

wynik = wynik & ~(maska << przesuniecie);

$$\begin{array}{rcl} 1011\ 0101\ \&\ \sim\ 00100000 & \\ \hline 1011\ 0101\ \&\ 11011111 & \longrightarrow \text{AND} \begin{array}{r} 1011\ 0101 \\ 1101\ 1111 \\ \hline 1001\ 0101 \end{array} \end{array}$$

Przesunięcie zer w masce na odpowiednie miejsce i wykonanie operacji **AND** ustawia odpowiednie bity na 0.

wynik &= ~(maska << przesuniecie);

Operator XOR

Chcemy odwrócić stan czterech najstarszych bitów w bajcie nie modyfikując pozostałych bitów.

```
wynik = wynik ^ maska;
```

```
wynik = 0b10101010;
```

```
maska = 0b11110000;
```

```
-----
```

```
XOR = 0b01011010;
```

```
wynik ^= maska;
```

Parę przykładów

```
% ustawiamy wszystkie bity na 1
```

```
DDRC = 0xFF;
```

```
% ustawiamy bit 7, 6 i 1 portu A jako wyjściowe, a pozostałe jako wejściowe
```

```
DDRA = (1<<7) | (1<<6) | (1<<1);
```

```
albo
```

```
DDRA = (1<<PA7) | (1<<PA6) | (1<<PA1);
```

```
% ustawiamy bit 7 i 0 portu A, nie modyfikujemy innych
```

```
PORTA |= ((1<<PA7) | (1<<PA0));
```

```
albo
```

```
PORTA |= ((1<<7) | (1<<0));
```

```
% odczytujemy 4 najbardziej znaczące bity portu A
```

```
data = ( PINA & 0xF0 ) >> 4;
```

```
albo
```

```
data = ( PINA & ((1<<PA7)|(1<<PA6)|(1<<PA5)|(1<<PA4)) ) >> 4;
```

```
% odwracamy stan bitu nr 3
```

```
PORTA ^= 0x08;
```

```
albo
```

```
PORTA ^= (1<<PA3);
```

```
% sprawdzamy czy bit 6 jest ustawiony / wyzerowany
```

```
if (PINA & (1<<PA6))
```

```
if (!(PINA & (1<<PA6)))
```

```
% zadanie domowe
```

```
x = 0b00000011;
```

```
x = (x<<1 | x>>3);
```

Parę uwag

```
#define BIT(x) (1<<(x))
```

```
. . .
```

```
PORTA |= BIT(4);
```

Większość rejestrów ustawiamy tak:

```
TCCRO = (1<<FOC0) | (1<<WGM00) | (1<<CS00);
```

Jednak niektóre logiczniej jest ustawiać poprzez liczbę, np.:

Output Compare
Register – OCR0

Bit	7	6	5	4	3	2	1	0	
	OCR0[7:0]								OCR0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Output Compare Register contains an 8-bit value that is continuously compared with the counter value (TCNT0). A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC0 pin.

```
OCR0 = 0x53;
```

Parę uwag

Używając predefiniowanych w bibliotece io.h nazw sprawiamy, że kod jest niemal natychmiast kompatybilny z innymi mikrokontrolerami AVR.

```
// kod dla mega48
```

```
void spi_init(void){  
    DDRB |= (1 << PB2) | (1 << PB3) | (1 << PB5); //Turn on SS, MOSI, SCLK  
    SPCR=(1<<SPE) | (1<<MSTR); //enbl SPI, MSB 1st, init clk as low  
    SPSR=(1<<SPI2X);           //SPI at 2x speed (0.5 MHz)  
} //spi_init
```

```
// kod dla mega128
```

```
void spi_init(void){  
    DDRB |= (1<<PB2) | (1<<PB1) | (1<<PB0); //Turn on SS, MOSI, SCLK  
    SPCR=(1<<SPE) | (1<<MSTR); //enbl SPI, clk low init, rising edge sample  
    SPSR=(1<<SPI2X);           //SPI at 2x speed (8 MHz)  
} //spi_init
```

Widoczność zmiennych w oddzielnych plikach

File1.c

```
// zmienna globalna  
int count = 0;
```

Zmienna “count” jest widoczna we wszystkich plikach w projekcie.

File2.c

```
extern int count;  
int x = count;
```

Tak należy użyć zmiennej globalnej “count” zadeklarowanej w pliku file1.c.

Deklarację “extern” umieszcza się zwykle w pliku nagłówkowym *.h.

Widoczność zmiennych w oddzielnych plikach

File1.c

```
// zmienna globalna  
int count = 0;
```

Teraz chcemy użyć nazwy „count”
w wielu plikach, w każdym jako
niezależna zmienna.

File2.c

```
// inna zmienna  
// o tej samej nazwie  
int count = 100;
```

Kompilator zaprotestuje.

Widoczność zmiennych w oddzielnych plikach

File1.c

```
// zmienna globalna  
static int count = 0;
```

Poza funkcjami, specyfikator „static” ogranicza widoczność zmiennej „count” do tego pliku.

File2.c

```
// inna zmienna  
// o tej samej nazwie  
static int count = 100;
```

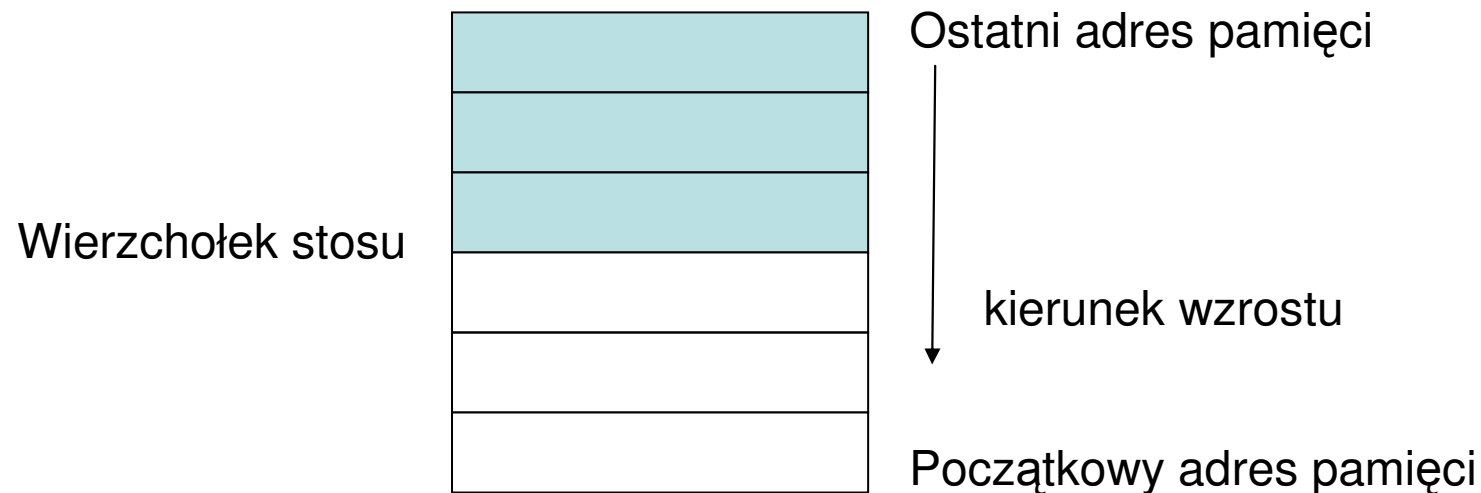
Powyższa zmienna „count” widoczna tylko pliku File2.c.

Specyfikator static **wewnątrz funkcji** sprawia, że zmienna jest umieszczana w tej samej pamięci, co zmienna globalna i nie jest usuwana wraz z zakończeniem funkcji.

```
int licznik()  
{  
    static int a;  
    a++;  
    return a;  
}
```

Funkcje i stos

Podstawowym zastosowaniem stosu jest zapamiętywanie adresów powrotu podczas wywoływania procedur. Stos wykorzystywany jest też jako rodzaj podręcznej pamięci do chwilowego przechowywania danych.



Na stosie mogą się znaleźć

- Adresy powrotu z funkcji
- Wartości zwracane przez funkcję
- Zmienne lokalne
- Zapisane wartości rejestrów

Przykład

```
void doNothing() {  
    char c;  
}  
int main() {  
    char x, y, z;  
    int i;  
    for (i = 0; i < 10; i++) {  
        doNothing();  
    }  
    return 0;  
}
```

1000	
999	
998	
997	
996	
995	
994	
993	

Przykład

```
void doNothing() {  
    char c;  
}  
int main() {  
    char x, y, z;  
    int i;  
    for (i = 0; i < 10; i++) {  
        doNothing();  
    }  
    return 0;  
}
```

1000	x
999	y
998	z
997	
996	
995	
994	
993	

Przykład

```
void doNothing() {  
    char c;  
}  
int main() {  
    char x, y, z;  
    int i;  
    for (i = 0; i < 10; i++) {  
        doNothing();  
    }  
    return 0;  
}
```

1000	x
999	y
998	z
997	iL
996	iH
995	
994	
993	

Przykład

```
void doNothing() {  
    char c;  
}  
int main() {  
    char x, y, z;  
    int i;  
    for (i = 0; i < 10; i++) {  
        doNothing();  
    }  
    return 0;  
}
```

1000	x
999	y
998	z
997	iL
996	iH
995	adres
994	linii 9
993	

Przykład

```
void doNothing() {  
    char c;  
}  
int main() {  
    char x, y, z;  
    int i;  
    for (i = 0; i < 10; i++) {  
        doNothing();  
    }  
    return 0;  
}
```

1000	x
999	y
998	z
997	iL
996	iH
995	adres linii 9
994	
993	c

Przykład

```
void doNothing() {  
    char c;  
}  
int main() {  
    char x, y, z;  
    int i;  
    for (i = 0; i < 10; i++) {  
        doNothing();  
    }  
    return 0;  
}
```

1000	x
999	y
998	z
997	iL
996	iH
995	
994	
993	



Stos może zniszczyć zmienne w pamięci RAM.

Tłuste parametry przekazywać do funkcji przez adres lub wskaźnik.

Zrezygnować z funkcji rekurencyjnych.

Zostawić przynajmniej 20% wolnej pamięci RAM na potrzeby STOSU.

Zadeklarowałem globalnie `int a[400]` →

Size after:AVR Memory Usage

Device: atmega16

Program: 218 bytes (1.3% Full)
(.text + .data + .bootloader)

Data: 930 bytes (90.8% Full)
(.data + .bss + .noinit)

Pamięć FLASH

W pamięci FLASH wraz programem wykonywalnym przechowywane są wartości początkowe zmiennych, które takową posiadają.

Poszczególnych komórek pamięci Flash nie możemy modyfikować w czasie działania (stałe napisy, większe tablice ze stałą wartością)

```
#include <avr/pgmspace.h>
const uint8_t tablica[3] PROGMEM = {48, 25, 150};
char lancuch[] PROGMEM = „Witaj”;
```

Przedrostek "**const**" nie jest wymagany, jednak jego obecność daje kompilatorowi wiadomość, że dana zmienna nie może być modyfikowana i w przypadku takiej próby wystąpi błąd kompilacji.

Pamięć FLASH

By móc odczytać stałą zapisaną w pamięci korzystamy z jednej z poniższych funkcji:

`pgm_read_byte(address)` ; - funkcja zwraca wartość stałej 8-bitowej
`pgm_read_word(address)` ; - funkcja zwraca wartość stałej 16-bitowej
`pgm_read_dword(address)` ; - funkcja zwraca wartość stałej 32-bitowej
`pgm_read_float(address)` ; - funkcja zwraca wartość stałej typu float.

A więc, jeżeli chcieli byśmy wczytać liczbę z naszej tablicy do zmiennej to napisalibyśmy:

```
uint8_t liczba = pgm_read_byte(&tablica[2]);
```

jeżeli zaś mamy do wczytania element łańcuchu możemy wczytać go podobnie jak wyżej:

```
char literka = pgm_read_byte(&lancuch[1]);
```

Pamięć EEPROM

Do EEPROM-u odnosimy się poprzez funkcje zadeklarowane w bibliotece **avr/eeprom.h**.

W przeciwieństwie do pamięci Flash zmienne zapisane w EEPROM-ie mogą być odczytywane jak i zapisywane.

EEPROM to pamięć typu nieulotnego.

Ograniczona liczba cykli zapisu (w przypadku tych wbudowanych w mikrokontrolery AVR producent gwarantuje 100 tysięcy poprawnie przeprowadzonych zapisów).

Pamięć tego typu stosujemy głównie do zapisania parametrów nastaw urządzenia, lub np. rzadko aktualizowanych, lub przeznaczonych przede wszystkim na odczyt zmiennych.

Mikrokontrolery AVR ATMEGA16 mają wbudowaną pamięć EEPROM o rozmiarze 512 bajtów.

Pamięć EEPROM

Zmienne deklarujemy tak:

```
uint8_t zmienna EEMEM = 128;
```

By skorzystać z pamięci EEPROM używamy funkcji:

`eeeprom_write_byte (*adr, val)` - zapisuje wartość val pod adres adr.

`eeeprom_read_byte (*adr)` - czyta zawartość pamięci pod adresem adr.

`eeeprom_read_word (*adr)` - czyta 16 bitową zawartość pamięci pod adresem adr.

`eeeprom_read_block (*buf, *adr, n)` - czyta n bajtów od adresu adr i zapisuje do pamięci SRAM w miejscu wskazywanym przez argument *buf.

A więc odczyt pamięci EEPROM wyglądał by np. tak:

```
uint8_t wartosc = eeeprom_read_byte(&zmienna);
```

a zapis:

```
eeeprom_write_byte(&zmienna, wartosc);
```

Przerwania

W odpowiedzi na określony sygnał mikrokontroler zawiesza chwilowo wykonywanie programu głównego i wykonuje **procedurę obsługi przerwania**. Po zakończeniu tej procedury mikrokontroler wraca do wykonywania programu głównego, począwszy od miejsca, w którym zostało ono zawieszone.

```
ISR (nazwa_przerwania)
{
    // ciało procedury obsługi przerwania
}
```

```
/* Interrupt vectors */
/* Vector 0 is the reset vector. */

/* External Interrupt Request 0 */
#define INT0_vect                _VECTOR(1)

/* External Interrupt Request 1 */
#define INT1_vect                _VECTOR(2)

/* Timer/Counter1 Compare Match B */
#define TIMER1_COMPB_vect       _VECTOR(7)

/* ADC Conversion Complete */
#define ADC_vect                 _VECTOR(14)

/* 2-wire Serial Interface */
#define TWI_vect                 _VECTOR(17)
```

iom16.h

Specyfikator volatile

Zawsze, gdy chcemy, by kompilator nie optymalizował dostępu do takiej zmiennej.

Optymalizacja polega na tym, że po wejściu do funkcji kompilator zapamiętuje sobie zawartość komórki tej pamięci w wolnym rejestrze mikrokontrolera. Potem operuje tylko na tym rejestrze, aż do wyjścia z funkcji. Potem aktualizacja komórki pamięci.

Pożyteczne, bo szybsze.

Ale co, jeśli przerwanie ma za zadanie wykonać operację na tej samej zmiennej?

```
volatile uint8_t przycisk;

ISR (TIMER3_COMPA_vect)
{
    przycisk = ...;
}

main()
{
    while (!przycisk) {}
    ... // dalszy kod
}
```


Pamiętajmy, że rejestry są 8-bitowe. Wykorzystując zmienną dwubajtową w głównym kodzie i w przerwaniu musimy się zabezpieczyć.

```
//...
volatile uint16_t licznik16bit;
//...

ISR(...)
{
    licznik16bit++;
}

int main(void)
{
    uint16_t tmp;

    tmp = licznik16bit;    // źle. Przerwanie może nastąpić
                          // między przepisaniem niższego
                          // i wyższego bajtu

    cli(); // Wyłączamy przerwania
    tmp = licznik16bit;
    sei(); // Włączamy przerwania
```

Zmiana danego rejestru w głównym programie i w procedurze przerwania

```
#include <avr/io.h>
int main(void)
{
    //...
    PORTA |= (1<<PA0);
    PORTA |= (1<<PA2) | (1<<PA3) | (1<<PA4);
    //...
}
```

```
PORTA |= (1<<PA0);
d2: d8 9a      sbi 0x1b, 0

PORTA |= (1<<PA2) | (1<<PA3) | (1<<PA4);
d4: 8b b3      in r24, 0x1b
d6: 8c 61      ori r24, 0x1C
d8: 8b bb      out 0x1b, r24
...
```

Ustaw bit 0 komórki o adresie 0x1b (PORTA)

Wczytaj do rejestru 24 wartość komórki o adresie 1b

Tu przerwanie zeruje bit 0 portu A

Dodaj logicznie do r24 0b00011100

Przepisz do portu A wartość rejestru

W rezultacie nie ma śladu po przerwaniu...

Zawsze tak pisz kod programu, jak gdyby gość, który z niego korzysta w pracy, był agresywnym psychopata wiedzącym, gdzie mieszkasz.

Damian Conway