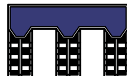
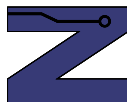
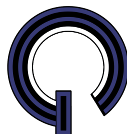
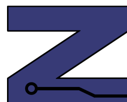
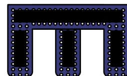
Co-funded by the
Erasmus+ Programme
of the European Union

"Teaching online electronics, microcontrollers and programming in Higher Education"
Erasmus+ Programme
2020-1-PL01-KA226-HE-095653

Podstawy systemów mikroprocesorowych: Jezyk C dla systemów mikroprocesorowych

dr inż. Dariusz Tefelski

dariusz.tefelski@pw.edu.pl

Pokój 225GF

Dotychczas program wyglądał mniej więcej tak:

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

Odtąd powinien wyglądać tak:

```
#include <avr/io.h>
int main(void)
{
    while(1)
    {
    }
    return 0;
}
```

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;
    while(1){
        PORTD^=0xff; //migamy
        _delay_ms(CZAS);
    }
    return 0;
}
```

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

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

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

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;
    while(1){
        PORTD^=0xff; //migamy
        _delay_ms(CZAS);
    }
    return 0;
}
```

Definicje

- Dyrektywa **#define**.
- Makra preprocesora. Ich argumenty podstawiane do kodu przed kompilacją. Dobry nawyk - drukowana czcionka.

```
#define F_CPU 16000000 //szybkość CPU w Hz

#define LICZ(a,b) a+b

a = 2 * LICZ(5,2); -> a = 2 * 5 + 2=12;

#define LICZ2(a,b) (a+b)

a = 2 * LICZ2(5,2); -> a = 2 * (5 + 2)=14;

#define LED_DDR DDRC
#define LED PC0
LED_DDR|=(1<<LED);
```

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;
    while(1){
        PORTD^=0xff; //migamy
        _delay_ms(CZAS);
    }
    return 0;
}
```

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 int64_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) //ZAWSZE
{
DDRD=0xff;
while(1){
PORTD^=0xff; //migamy
_delay_ms(CZAS);
}
return 0;
}
```

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; //USTAWIAMY REJESTRY
while(1){
PORTD^=0xff; //migamy
_delay_ms(CZAS);
}
return 0;
}
```

Plik 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_ATmega649P__)
#include <avr/iom649p.h>
#elif defined (__AVR_ATmega64HVE__)
#include <avr/iom64hve.h>
#elif defined (__AVR_ATmega103__)
#include <avr/iom103.h>
#elif defined (__AVR_ATmega32__)
#include <avr/iom32.h>
#elif defined (__AVR_ATmega323__)
#include <avr/iom323.h>
```

Plik iom32.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

/* UCSRA */
#define UCSRA          _SFR_IO8(0x0B)
#define RXC  7
#define TXC  6
#define UDRE 5
#define FE   4
#define DOR  3
#define PE   2
#define U2X  1
#define MPCM 0
```

Rejestry DDR, PORT oraz PIN

- Rejestry DDR służą do ustalenia roli pinu (wejściowy/wyjściowy).
- Aby dany pin pełnił rolę pinu wyjściowego, należy ustawić wartość 1 na danym pinie w rejestrze DDR.
- Aby dany pin pełnił rolę pinu wejściowego, należy ustawić wartość 0 na danym pinie w rejestrze DDR.

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	

Przykład 1:
Ustawienie wszystkich pinów portu A jako wyjściowych:

```
DDRA = 0xff;
//albo
DDRA = 0b11111111; //0b oznacza zapis binarny
```

Rejestry DDR, PORT oraz PIN

- Rejestr PORT służy do ustawiania wybranej wartości (0 = 0V bądź 1=5V) na danym pinie.

Port A Data Register – PORTA

Bit	7	6	5	4	3	2	1	0
	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
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

- Rejestr PIN służy do odczytywania wartości z danego pinu.

Port A Input Pins Address – PINA

Bit	7	6	5	4	3	2	1	0
	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
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

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

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

Operator AND

Przykład nr 1:
Zmiana bitu nr 4 na 0:

```
char x= 0b10111100;
char y= 0b11101111; //wykorzystamy y jako maskę
char z;
z = x & y;
```

Maska i operator & zmieniają stan bitu na 0.

Operator OR

Przykład nr 2:
Zmiana bitu nr 4 na 1:

```
char x = 0b10000101;
char y = 0b00010000;
char z;
z = x | y;
```

Maska i operator | zmieniają stan bitu na 1.

Operator SHIFT

```
char x = 0b 0000 0001;
char y = 5;
char z;

z = x << 5; //z = 0b 00100000
```

- Operator $<<$ przesuwa x o y miejsc w lewo dopisując y zer na sam koniec.
- Operator $>>$ przesuwa x o y miejsc w prawo dopisując y zer z przodu.

Operator OR i SHIFT

Powróćmy do przykładu nr 2: Zmiana bitu nr 4 na 1:

```
char x = 0b10000101;
char maska = 0b 0000 0001;
char przesuniecie = 4;
x = x | (maska << przesuniecie);
```

Co się dzieje po kolei:

1. Przesunięcie maski o 4 pozycje w lewo:
   0b 0000 0001 → 0b 0001 0000
2. Wykonanie operacji OR pomiędzy wartością x a rezultatem z poprzedniego punktu:
   0b 1000 0101
   | 0b 0001 0000
   ―――――――――
   → 0b 1001 0101

Alternatywne zapisy:

```
x|=(maska<<przesuniecie);
x|=(1<<przesuniecie);
```

Operator AND i SHIFT

Powróćmy do przykładu nr 1: Zmiana bitu nr 4 na 0:

```
char x= 0b101 1 1100;
char maska = 0b0000 0001;
char przesuniecie=4;
x = x & ~(maska<<przesuniecie);
```

Co się dzieje po kolei:

1. Przesunięcie maski o 4 pozycje w lewo:
   0b 0000 0001 → 0b 0001 0000
2. Negacja wyniku z punktu 1:
   0b 0001 0000 → 0b 1110 1111
3. Wykonanie operacji AND pomiędzy wartością x a rezultatem z punktu 2:
   0b 1011 1100
   & 0b 1110 1111

   → 0b 1010 1100

   Alternatywne zapisy:

   ```
   x &= ~(maska << przesuniecie);
   x &= ~(1 << przesuniecie);
   ```

Operator XOR

- Chcemy odwrócić stan czterech najstarszych bitów w bajcie nie modyfikując pozostałych bitów:

  wynik = wynik ^ maska;

  wynik = 0b10101010;
  maska = 0b11110000;

  XOR = 0b01011010

- Alternatywny zapis: 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);
```

Parę przykładów

```
//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)))
```

Zapamiętaj!

- Ustawienie bitu: PORTD |= (1 << numer pinu);
- Wyzerowanie bitu: PORTC & = ∼ (1 << numer pinu);
- Sprawdzenie bitu: if(!(PINA & (1 << numer pinu)))

```
#define USTAWBIT(x) (1<<x)
...
PORTA |=USTAWBIT(4);
```

Większość rejestrów ustawiamy tak:

```
TCCR0 = (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
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
SPSR=(1<<SPI2X);
//SPI at 2x speed (8 MHz)
}//spi_init
```

Widoczność zmiennych w oddzielnych plikach

Plik file1.c

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

Zmienna globalna "count" normalnie nie jest widoczna we wszystkich plikach w projekcie, tzn. nie można jej użyć, bez wcześniejszej deklaracji.

Plik 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

Plik File1.c

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

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

Plik File2.c

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

Kompilator zaprotestuje.

Widoczność zmiennych w oddzielnych plikach

Plik File1.c

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

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

Plik File2.c

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

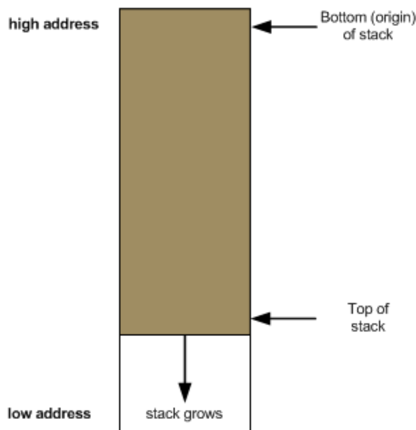
Powyższa zmienna ”count” widoczna tylko pliku File2.c.

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.



Source: `https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/`

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;
    }
```

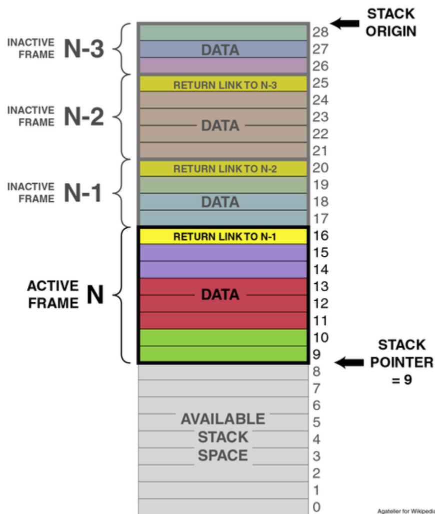
1000	X
999	y
998	z
997	iL
996	iH
995	adres
994	linii 9
993	

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
994	linii 9
993	c



Zadeklarowałem globalnie:
int a[400]; →

- Stos może zniszczyć zmienne w pamięci RAM.
- *Tłuste* parametry przekazywać do funkcji przez adres lub wskaźnik.
- Zrezygnować z funkcji rekurencyjnych.
- Zostawić przynajmniej 20wolnej pamięci RAM na potrzeby STOSU.

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 takowe 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 chcielibyśmy 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 govpodobnie 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 ATMEGA32 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:

```
eeprom_write_byte ( *adr, val) // zapisuje wartość val
// pod adres adr.
eeprom_read_byte ( *adr ) // czyta zawartość pamięci pod adr.
eeprom_read_word ( *adr ) // czyta 16 bitową zawartość pamięci
// pod adresem adr.
eeprom_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 = eeprom_read_byte(&zmienna);
```

a zapis:

```
eeprom_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.

Plik main.c

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

Plik iom32

```
/* 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)
```

Specyfikator volatile

- Zawsze, gdy chcemy, by kompilator nie optymalizował dostępu do 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?

```
uint8_t
volatile przycisk;
uint8_t przycisk;
ISR (TIMER3_COMPA_vect){
    przycisk = ...;
}
int main(void){
    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

   //Poprawnie:
   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);
    //...
}
```

Rezultat:

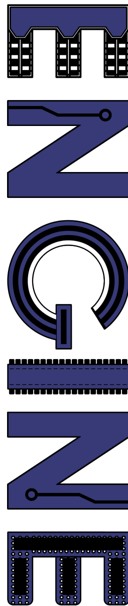
```
PORTA |= (1<<PA0);
d2: d8 9a     sbi 0x1b, 0 //Ustaw bit 0 komórki o adresie 0x1b
PORTA |= (1<<PA2)|(1<<PA3)|(1<<PA4);
d4: 8b b3     in r24, 0x1b //Wczytaj do rejestru 24 wartość
//komórki o adresie 1b
Tu przerwanie zeruje bit 0 portu A
d6: 8c 61     ori r24, 0x1C //Dodaj logicznie do r24 0b00011100
d8: 8b bb     out 0x1b, r24 //Przepisz do portu A wartość
...
```

W rezultacie nie ma śladu po przerwaniu...

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

Damian Conway

Dziękuję za uwagę.



Pierwotna wersja wykładu jest dziełem:
dr hab. Piotra Fronczaka, mgr inż. Marcina Zaręby.