Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

KIERUNEK STUDIÓW INFORMATYKA

MATERIAŁY DO ZAJĘĆ LABORATORYJNYCH

Systemy wbudowane

Dr inż. Wojciech Surtel:

Lublin 2020

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

LABORATORIUM 2. PROGRAM WBUDOWANY DLA MIKROKONTROLERA AVR. PRACA W ŚRODOWISKU IDE.

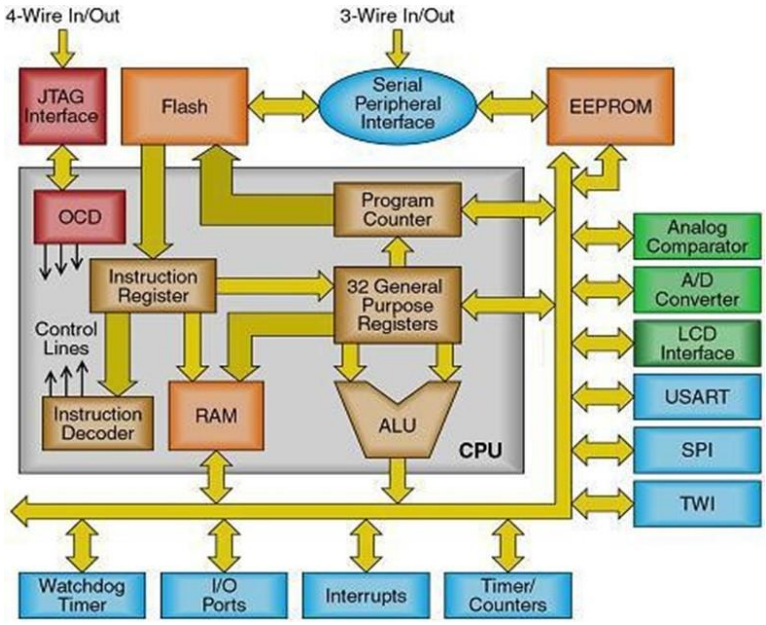
Cel laboratorium:

Wprowadzenie do programowania w języku C mikrokontrolerów rodziny AVR na przykładzie układu ATmega32.

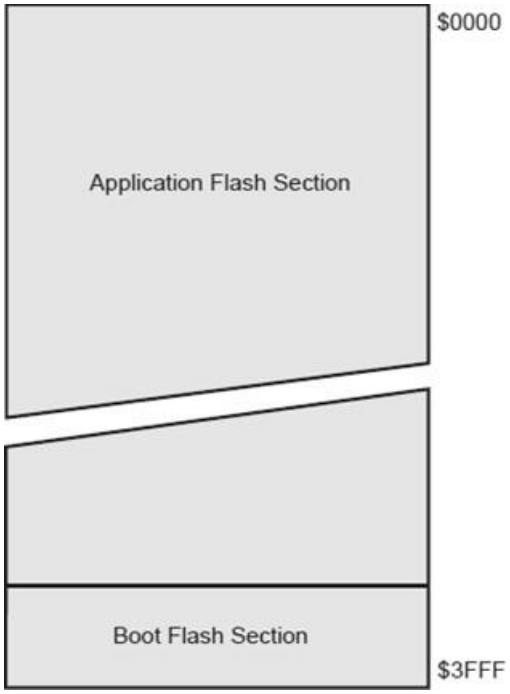
Zakres tematyczny zajęć:

zakres 1,

Mikrokontroler z serii AVR jest to 8 bitowy mikrokontroler typu RISC. Budowa jego opiera się o architekturę harwardzką. Zgodnie z tą architekturą w mikrokontrolerze rozdzielone są magistrale pamięci programu (16bitów) i pamięci danych (8bitów). Odpowiednia konfiguracja zewnętrzna mikrokontrolerów umożliwia dołączenie zewnętrznej pamięci danych o rozmiarze do 64 KB. Niestety brak jest możliwości dołożenia zewnętrznej pamięci programu. Dużą szybkość mikrokontrolera zapewnia przetwarzanie potokowe, powodujące wykonywanie większości rozkazów mieszczących się w jednym cyklu zegarowym, oraz 32 bajtowy obszar rejestrów roboczych, o natychmiastowym dostępie. Ich dodatkową zaletą jest brak ścisłego określenia akumulatora. Tę funkcję może pełnić dowolnie wybrany rejestr, spośród 32-bajtowego banku rejestrów roboczych. Zastosowanie szeregowego algorytmu programowania oraz pamięć programu typu "Flash", umożliwia programowanie i przeprogramowanie mikrokontrolera po umieszczeniu go w układzie. Konstruktorzy uwzględnili również układ Watchdog, jak i tryb pracy z obniżonym poborem mocy, które w obecnej chwili stają się standardem w budowie mikrokontrolerów. Poniżej na rysunkach przedstawiono architekturę jednostki centralnej; mapę pamięci i zestaw rejestrów mikrokontrolera.

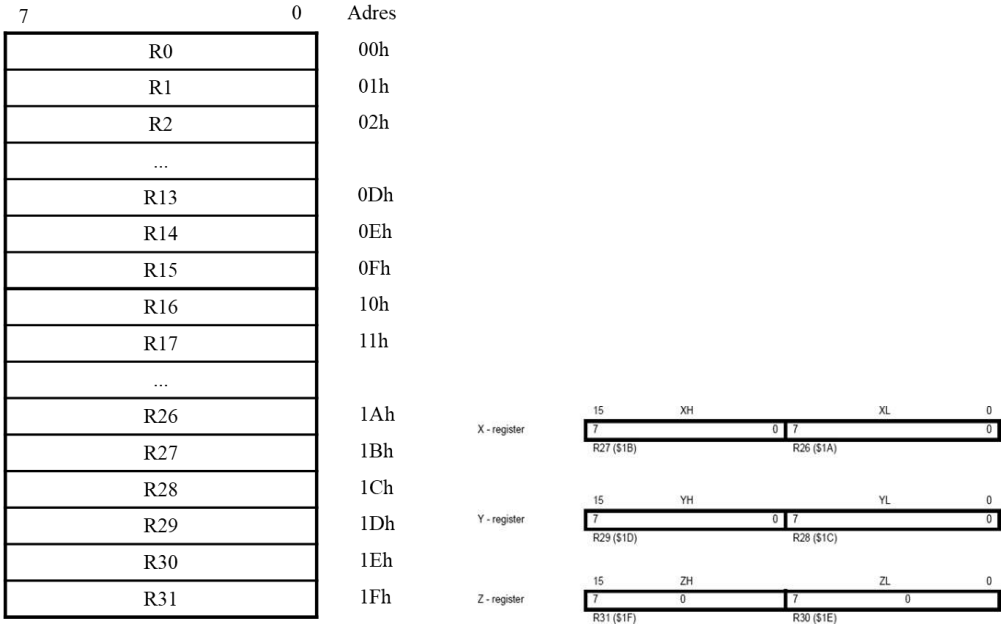


Rys. 2.1. Schemat blokowy mikrokontrolera z rdzeniem AVR z wyróżnioną jednostką centralną CPU



Rys. 2.2. Mapa pamięci mikrokontrolera ATmega32

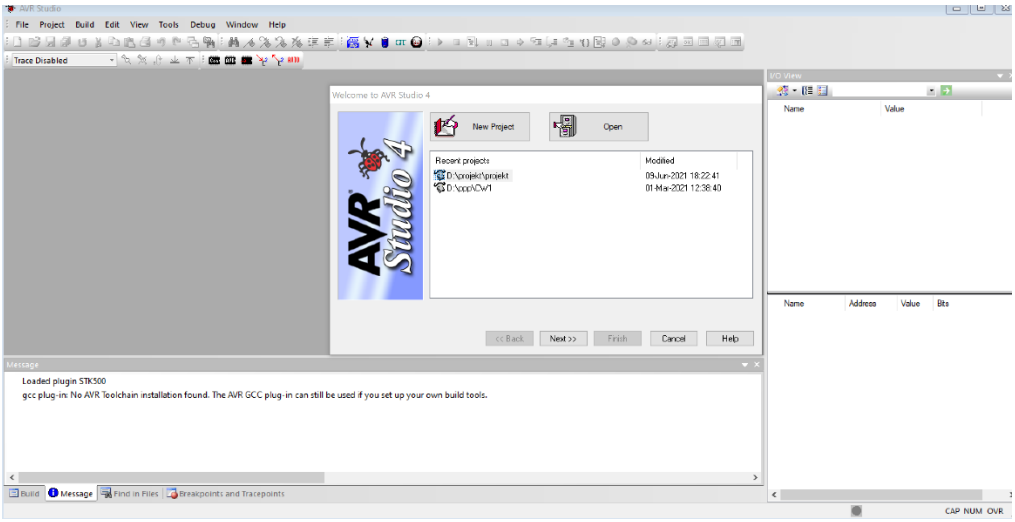
Mikrokontroler posiada 32kB pamięci Flesh do przechowywania programu o organizacji 16k × 16. Instrukcje AVR są 16 lub 32 bitowe. Licznik programu (PC) jest 14 bitowy, umożliwia on zaadresowanie 2^{14} = 16kB komórek pamięci. Pamięć programu jest podzielona na dwa obszary - sekcja aplikacji i sekcja bootloadera.



Rys. 2.3. Rejestry ogólnego przeznaczenia jednostki centralnej mikrokontrolera ATmega32

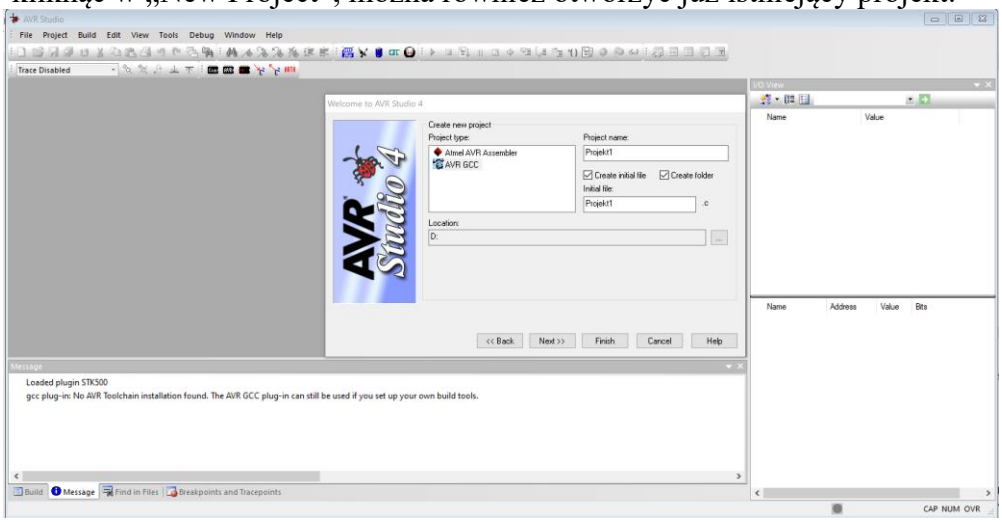
Rejestry X, Y i Z są to rejestry ogólnego przeznaczenia, dodatkowo służące do adresowania pośredniego w przestrzeni danych. Pamięć danych EEPROM mikrokontrolera AVR ATmega32 posiada następujące cechy:
ATmega32 posiada nieulotną pamięć danych o rozmiarze 1024 bajty, zorganizowanej w osobnej przestrzeni adresowej.
W przestrzeni tej mogą być zapisane lub odczytane pojedyncze bajty.
Dostęp do tej pamięci możliwy jest za pomocą rejestrów EEARH, EEARL, EEDR i EECR. Rejestry te pełnią następujące funkcje: EEARH i EEARL - adres, EEDR - wpisywane lub odczytywane dane, EECR - rejestr kontrolny.
Pamięć EEPROM data jest chroniona przed zniszczeniem, które może powodować zbyt niskie napięcie zasilania VCC.

zakres 2, tworzenie projektu dla dedykowanego środowiska programistycznego.



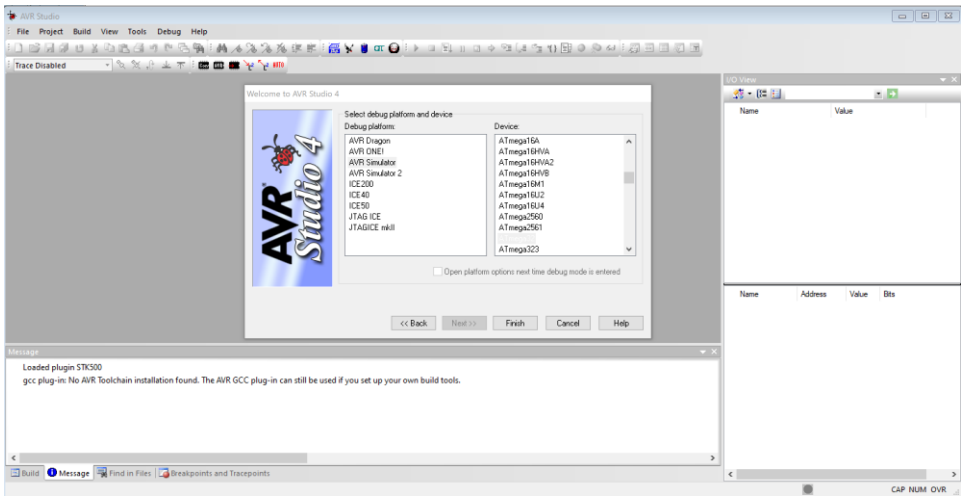
Rys. 2.4. Okno programu AvrStudio ver. 4

Po uruchomieniu programu, pojawia się następujące okienko. Aby stworzyć projekt należy kliknąć w „New Project”, można również otworzyć już istniejący projekt.



Rys. 2.5. Okno programu AvrStudio ver. 4., zakładanie nowego projektu

Wybranie odpowiedniego projektu i jego nazwy



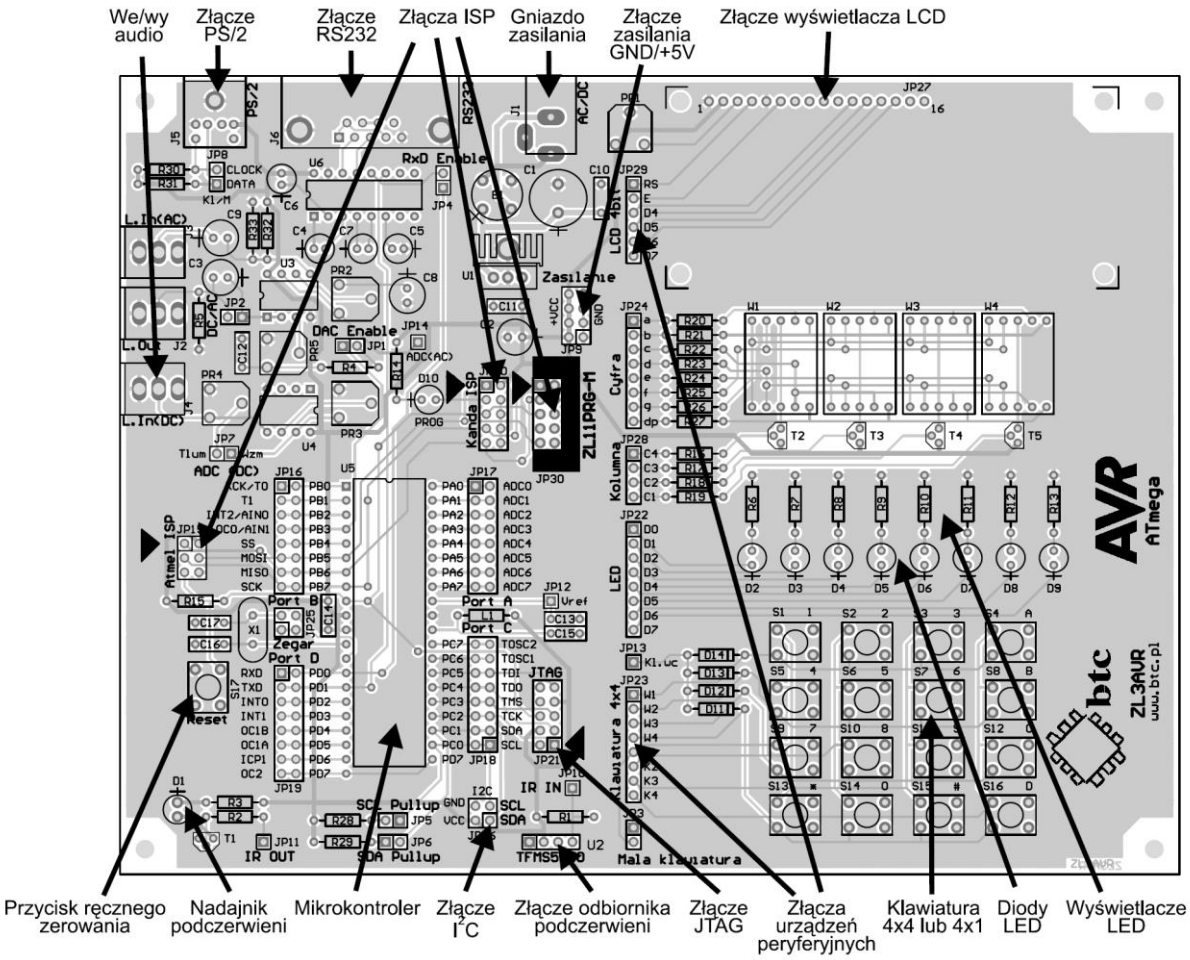
Rys. 2.6. Okno programu AvrStudio ver. 4., wybór platformy debugowania oraz typu mikrokontrolera

Wybór platformy do debugowania oraz urządzenia. Aby utworzyć pusty projekt należy kliknąć w „Finish”.

Skrypt programu wbudowanego dla kompilatora AVR-GCC wygląda następująco:

```
/* Szkielet przykładowego  programu dla avr-gcc*/
#define F_CPU 1000000L
/* Definiowanie stałych
 */
#include <avr/io.h>

/* Jeśli będą używanie funkcje w rodzaju _delay_ms, delay_us */
 #include <util/delay.h>

int main(void)
{
 /* Instrukcje - wstępne ustawienia, konfiguracje, inicjowanie itp. */
 /* Główna pętla programu */
     while(1)
     { /* Instrukcje w pętli */ }
}
```

Zobrazowanie chronologii tworzenia projektu przeprowadzimy na przykładzie zadania 1. Do realizacji zadania wykorzystamy zestaw edukacyjny ZL3AVR firmy BTC oraz środowisko AvrStudio ver. 4 z kompilatorem i Makefile – m z WinAVR ver. 20100110 wraz z nakładką emulatora układów peryferyjnych HAPSim.



Rys. 2.7. Rozmieszczenie układów oraz elementów zestawu ZL3AVR

Zadanie1.

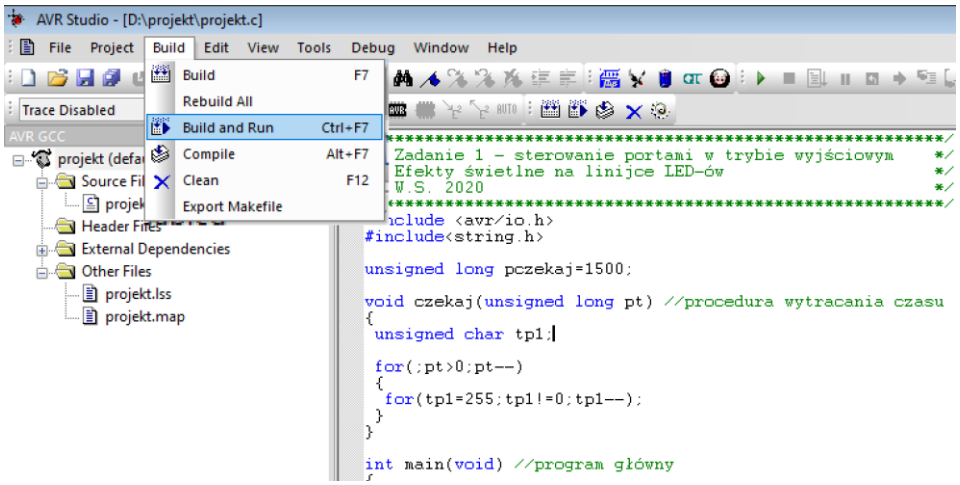
Sterowanie portami w trybie wyjściowym, efekty świetlne na linijce LED-ów Do portu B podłączone są diody LED (złącze JP22) w porządku: PB0 – D0, PB1 – D1, …, PB7 – D7.

KOD ZRÓDŁOWY:

```
/*************************************************************/
/* Zadanie 1 - sterowanie portami w trybie wyjściowym        */
/* Efekty świetlne na linijce LED-ów                         */
/* W.S. 2020                                                 */
/*************************************************************/
#include <avr/io.h>
#include<string.h>

unsigned long pczekaj=1500;

void czekaj(unsigned long pt) //procedura opóźnienia o zadany czas pt
{
```

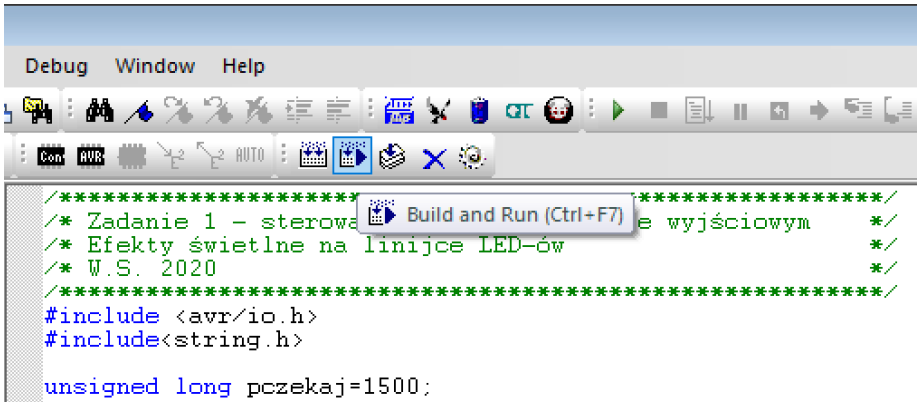
```
 unsigned char tp1;

 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
   {
    PORTB|=_BV(i);;   //wysteruj (zapal) pojedynczego LED-a
    czekaj(pczekaj);
   }
   for(i=0;i<8;i++)   //pętla zmieniająca fazę efektu
   {
   PORTB&=~_BV(i);    //wysteruj (zgaś) pojedynczego LED-a
    czekaj(pczekaj);  //opóźnij o zadany czas
   }
  }

//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);  //opóźnij o zadany czas
   }
  }
 }
}
```

Aby skompilować kod należy wybrać z menu głównego Build->Build and Run. Opcjonalnie można użyć skrótu Ctrl F7.



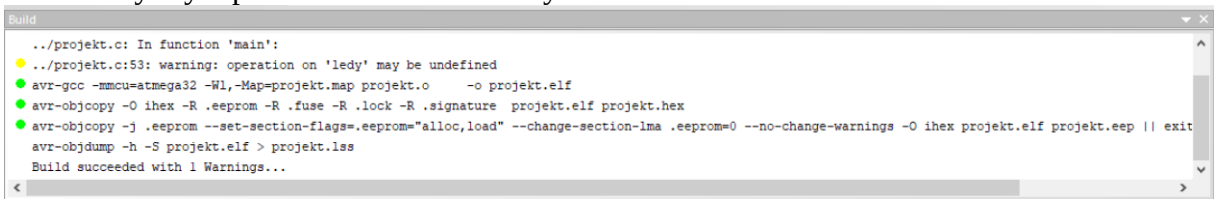
Rys. 2.8. Rozwinięcie zakładki Build w pasku narzędzi AVR Studio

Można również użyć ikony na pasku narzędzi.



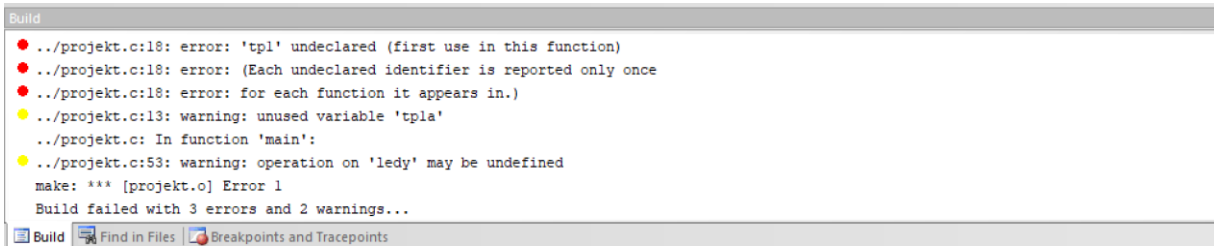
Rys. 2.9. Widok kontrolki Build and Run na pasku narzędzi AVR Studio

Po poprawnej asemblacji w oknie Build wyświetli się log z procesu. Jeżeli wszystko się udało zostaniemy i tym poinformowani stosownym komunikatem.



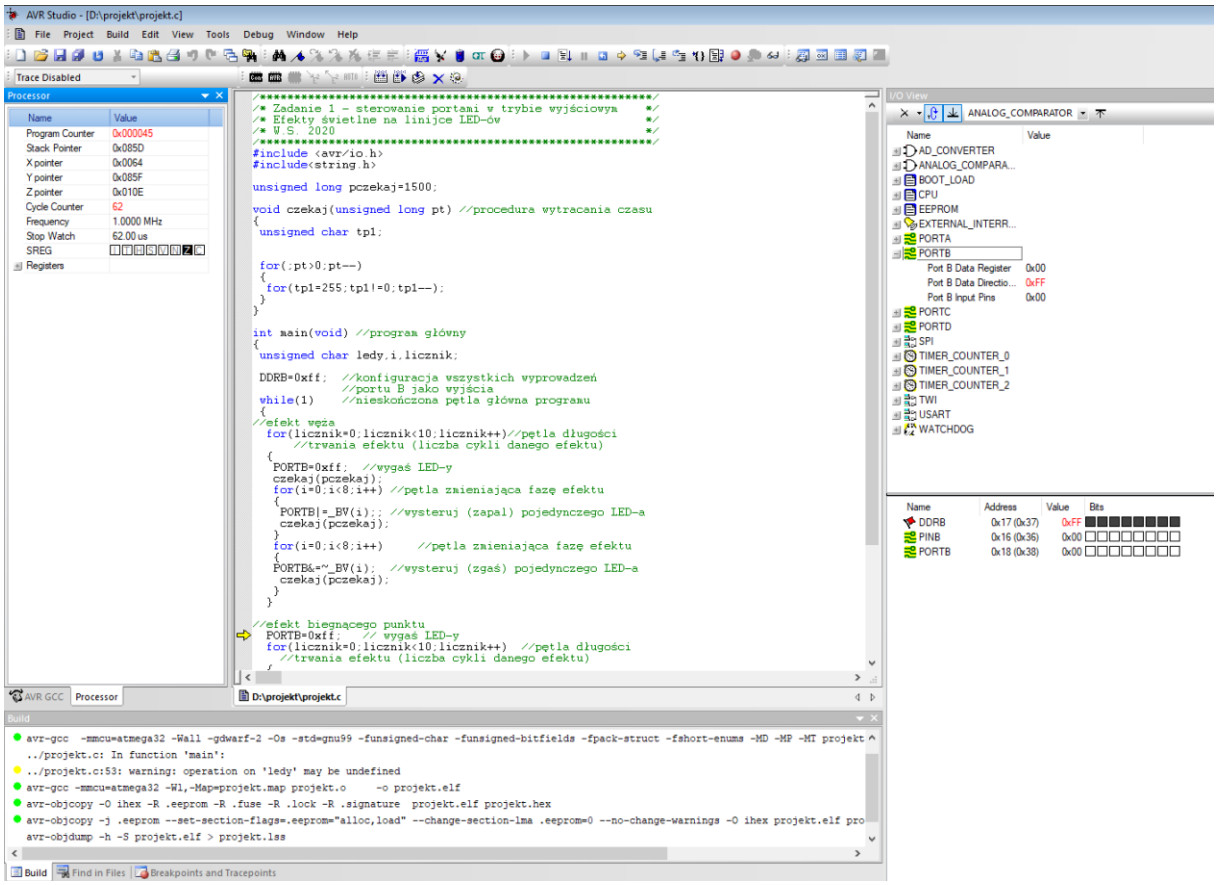
Rys. 2.10. Widok komunikatu w oknie Build dla poprawnego procesu kompilacji programu

W przypadku błędów w kodzie programu, w okienku Build pojawią się informacje na ich temat pomagając w lokalizacji błędów. Po dwukrotnym kliknięciu na daną pozycję w liście błędów kursor zostanie automatycznie przeniesiony w miejsce wystąpienie błędu.



Rys. 2.11. Widok komunikatu w oknie Build dla błędnego procesu kompilacji programu

Okno główne programu:

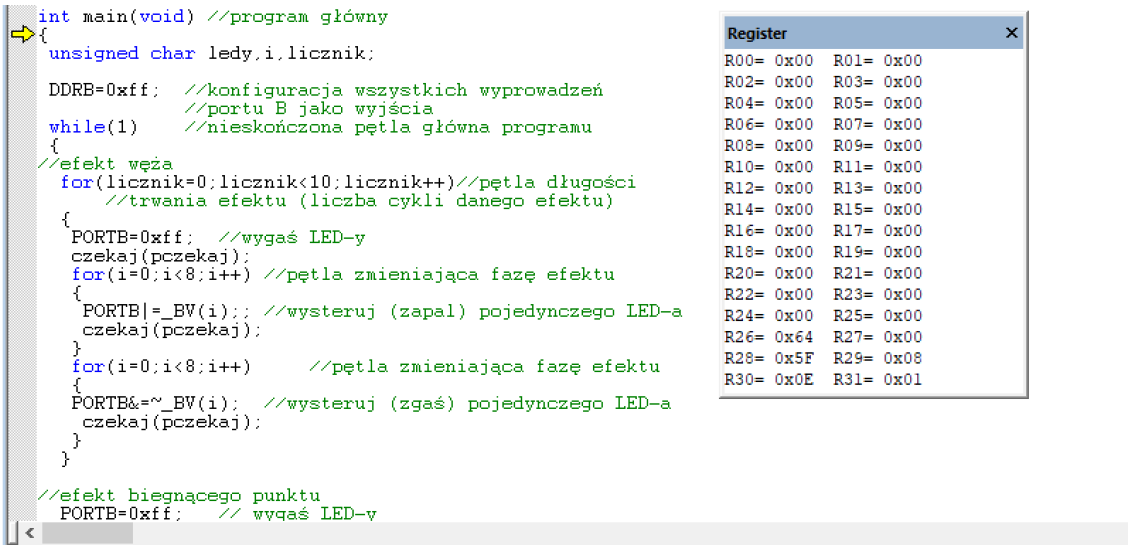


Rys. 2.12. Widok okna głównego programu AVR Studio ver. 4

Po pomyślnej kompilacji programu można przystąpić do debugowania. AVR Studio posiada bardzo wygodny i łatwy w obsłudze debugger.

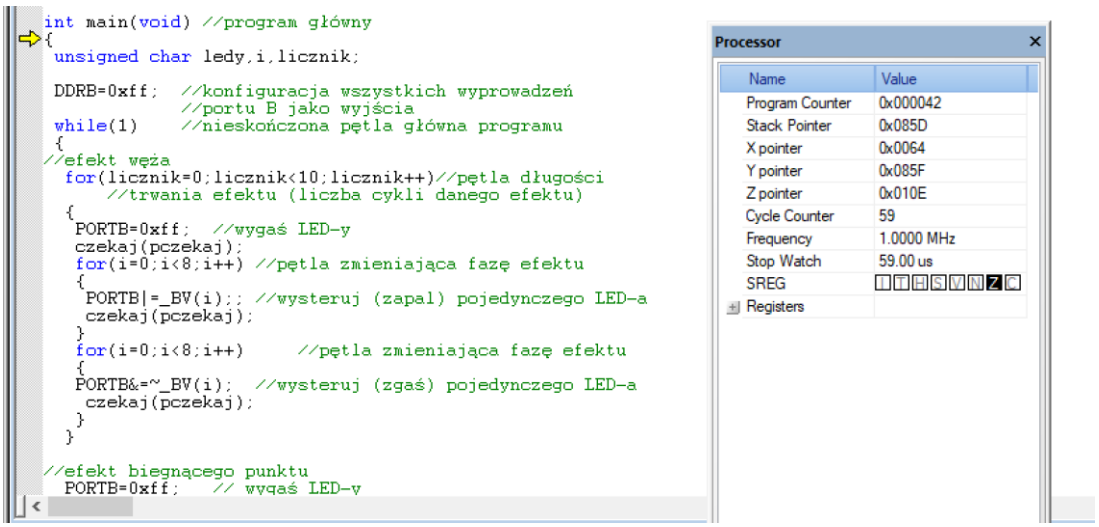
Debugowanie rozpoczynamy przyciskiem  , Debugger zatrzymuje się na pierwszej instrukcji.

W procesie debugowania bardzo przydatny jest I/O View, który pozwala na sprawdzenie stanów poszczególnych rejestrów procesora w dowolnym momencie:



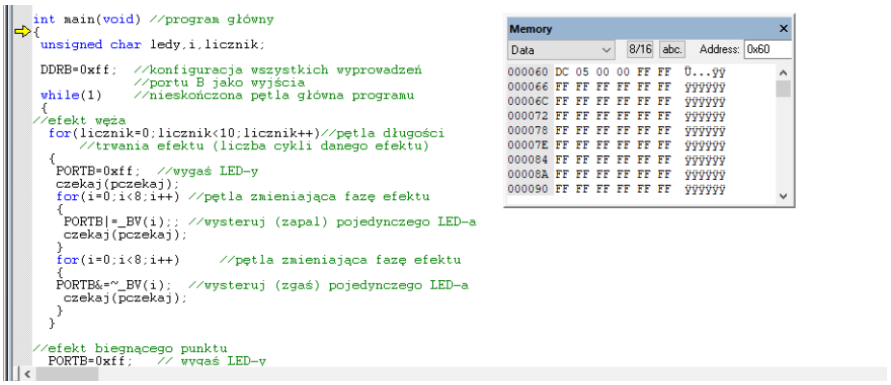
Rys. 2.13. Widok okna programu z podglądem rejestry procesora

Możemy również sprawdzić stan procesora poprzez podejrzenie licznika rozkazów, czy wskaźnika stosu:



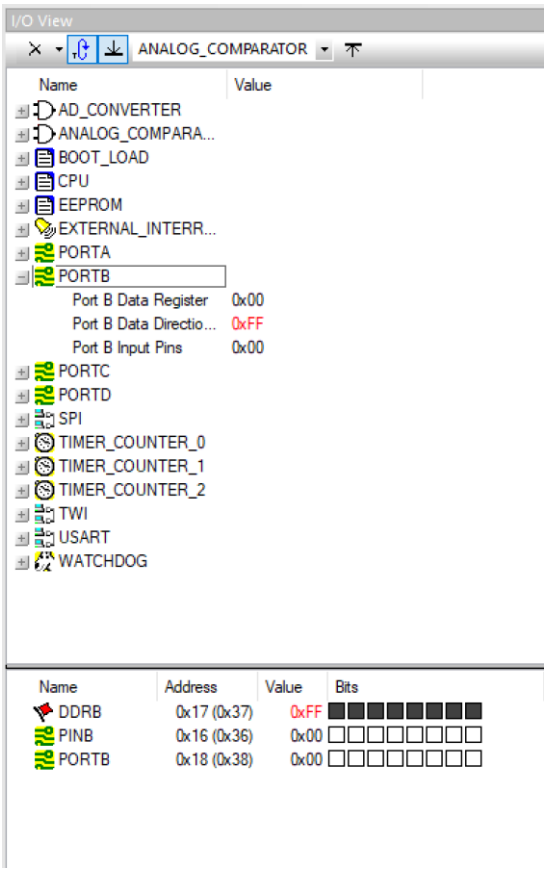
Rys. 2.14. Widok okna programu z podglądem stanu procesora

Dysponujemy także podglądem stosu, który organizowany jest w pamięci Data od adresu 0x60:



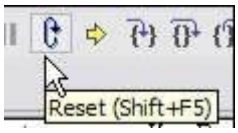
Rys. 2.15. Widok okna programu z podglądem stanu pamięci Data

Najbardziej przydatnym elementem debugger jest podgląd portów wejścia / wyjścia. Pozwala on po wstrzymaniu wykonania programu, obejrzeć zawartość lub dowolnie zmienić zawartość rejestrów (np. portu B).



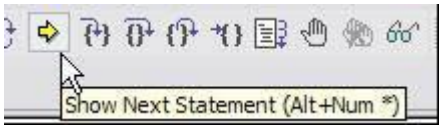
Rys. 2.16. Widok okna programu z podglądem na stan rejestrów specjalnych układów we/wy

Do manipulowania procesem debugowania służą następujące przyciski:



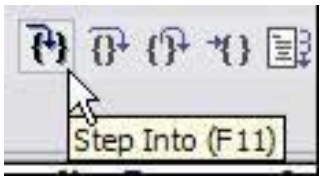
Rys. 2.17. Widok kontrolki Reset na pasku narzędzi AVR Studio

Reset służy do przerwania programu i uruchomienia go od nowa w dowolnym momencie.



Rys. 2.18. Widok kontrolki Show Next Statement na pasku narzędzi AVR Studio

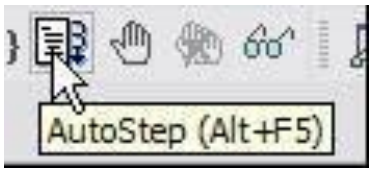
Przycisk Show Next Statement służy do przeniesienia kursora do aktualnie wykonywanej instrukcji programu.



Rys. 2.19. Widok kontrolek Step Into, Step Over, Step Out oraz Run To Cursor na pasku narzędzi AVR Studio

Step Into, Step Over, Step Out oraz Run To Cursor służą kolejno do:

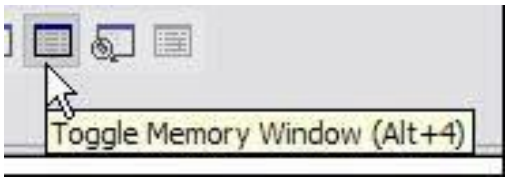
- przejścia do następnej instrukcji wraz z ewentualnym wejściem do podprogramu - przejścia do następnej instrukcji bez wchodzenia do podprogramu,
- wyjścia z podprogramu,
- uruchomienia programu i zatrzymaniu go w miejscu, w którym znajduje się kursor.



Rys. 2.20. Widok kontrolki Auto Step na pasku narzędzi AVR Studio

Auto Step automatycznie przechodzi do kolejnych instrukcji aż do zakończenia programu

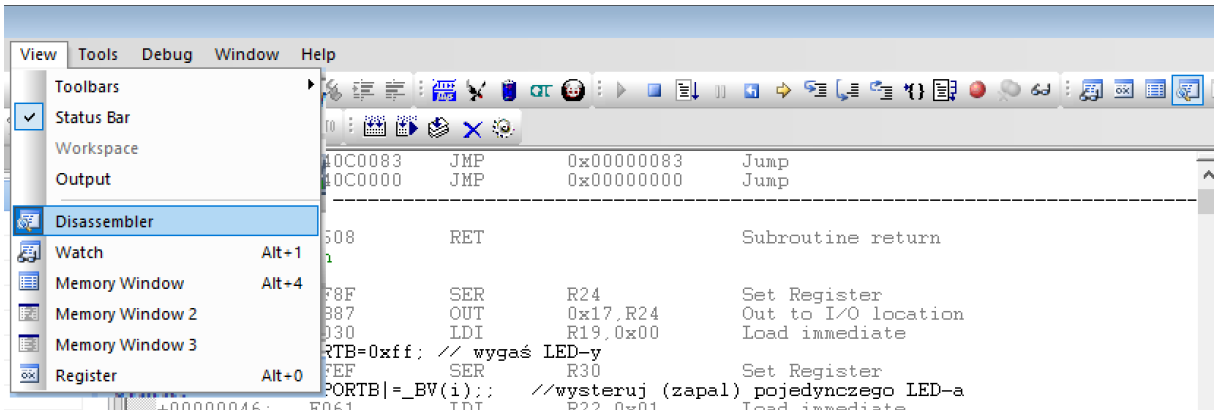
Kolejną przydatną opcją jest również memory window umożliwiające podgląd zawartości pamięci (Programu, ROM, I/O itp.)



Rys. 2.21. Widok kontrolki toggle memory window na pasku narzędzi AVR Studio

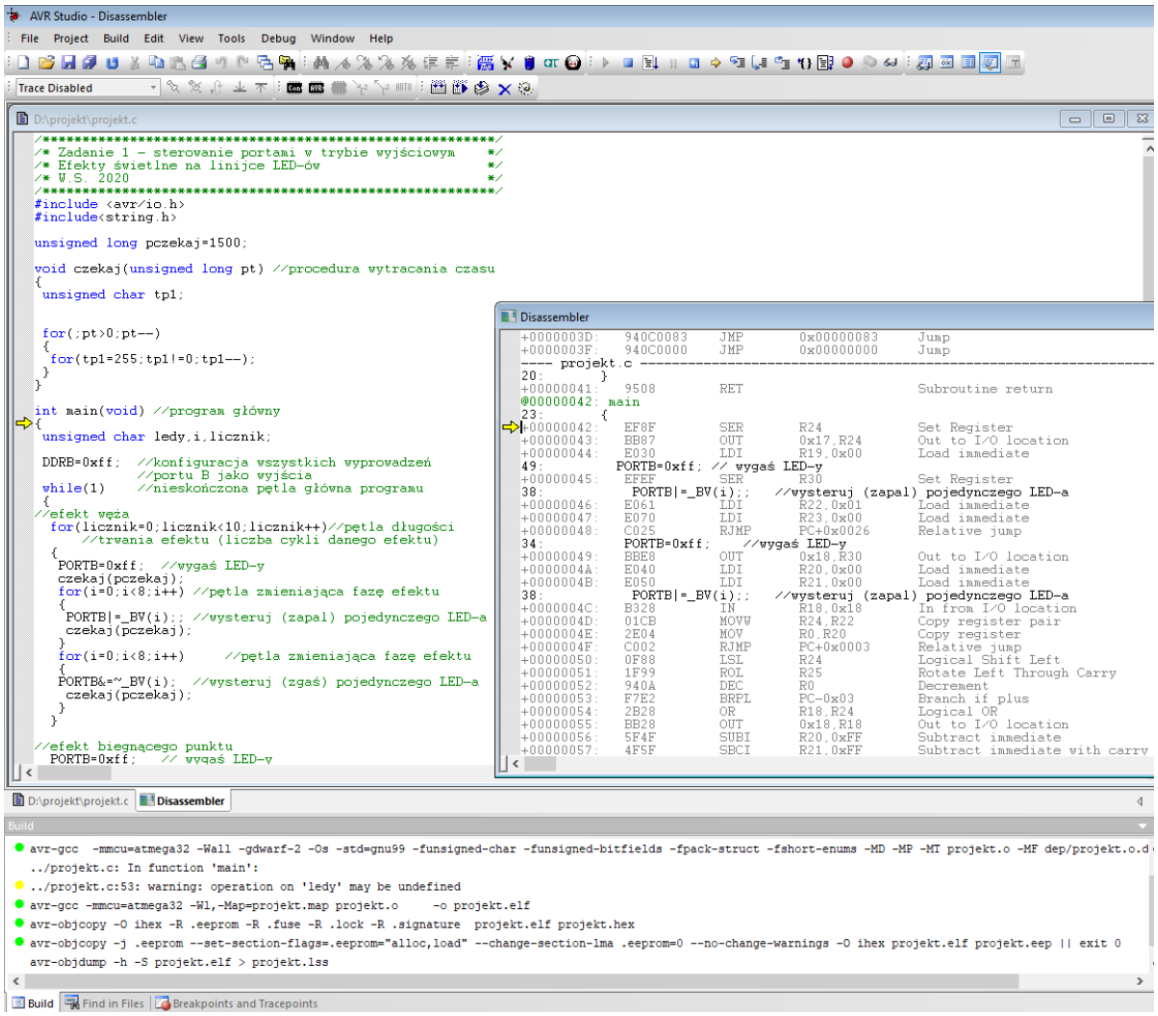
W analizie pracy systemu wbudowanego przydatnym narzędziem jest disassembler. Pozwala na podgląd pamięci programu. W przypadku pisania kodu w języku C, mamy możliwość sprawdzenia jakie zasoby CPU biorą udział w wykonaniu programu oraz jaka jest czasochłonność wykonania pojedynczych instrukcji języka C przez mikrokontroler liczona w liczbie rozkazów asemblerowych.

Wywołanie okna możliwe jest po zbudowaniu (ctrl + F7) Projektu. Wywołujemy je:



Rys. 1.22. Widok zakładki View ze wskazaniem na wybór Disassemblera

Na rys. 1.23 prezentuję okno disassemblera z podglądem na kod modułu main.



Rys. 1.23. Widok okna programu z wywołanym Disassemblerem

Pytania kontrolne:

1. Pytanie 1. Co to jest nagłówek programu? Zapisz w nagłówku konfigurację portu A pracującego jako: wszystkie linie w stan wyjścia.
2. Pytanie 2. Jak tworzymy pętlę główną programu?
3. Pytanie 3. Co to jest disassembler i jak go wywołać?

Zadanie 2.1 Modyfikacja zadania 1. Ćwiczenie z obsługi środowiska programistycznego.

Polecenie 1. Dołącz do Projektu bibliotekę delay.h. Zastąp procedurę czekaj(pczekaj); odpowiednią procedurą z biblioteki _delay_ms(int x); gdzie x – krotność opóźnienia 1 ms. Wykonaj odpowiednie symulacje działania programu w trybie krokowym oraz ciągłym.

Polecenie 2 W pętli zmieniającą fazę efektu świetlnego zmień kierunek sterowania diodami LED. Wykonaj odpowiednie symulacje działania programu w trybie krokowym oraz ciągłym.

LABORATORIUM 3. UKŁADY PERYFERYJNE MIKROKONTROLERA AVR. PORTY.

Cel laboratorium:

Zapoznanie studentów z budową i sposobem programowania układów we/wy mikrokontrolera z rdzeniem AVR. Budowa programu wbudowanego z elementami pętli licznikowych oraz instrukcjami warunkowmi. Operacje selektywne na rejestrach specjalnych.

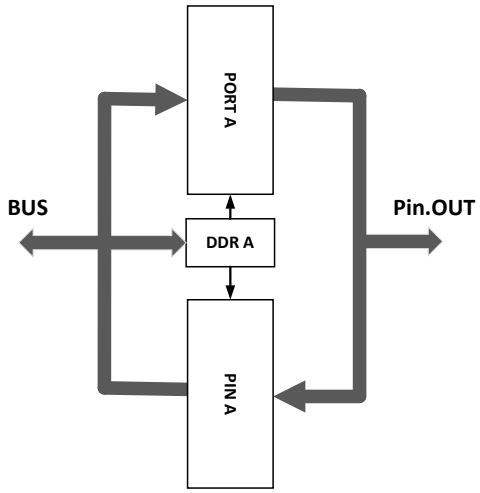
Zakres tematyczny zajęć:

zakres 1, Porty mikrokontrolera ATmega32.

Mikrokontroler ATmega32 zawiera 64 rejestry specjalne (rejestry I/O) i 32 rejestry uniwersalne (Rx). Nazwy rejestrów specjalnych są związane z funkcją jaką te rejestry pełnią, bądź ze sprzętem którego pracę konfigurują (np. rejestr statusu SREG, rejestry DDRx, PORTx, PINx portów I/O,). Natomiast nazwy rejestrów uniwersalnych składają się z numeru rejestru (od 0 do 31) poprzedzonych literą R (np. R16). Rejestry uniwersalne nie są dostępne z poziomu języka C. Dostęp do nich możliwy jest poprzez wstawki asemblerowe. Aby móc korzystać ze zdefiniowanych nazw rejestrów należy program rozpocząć dyrektywą:

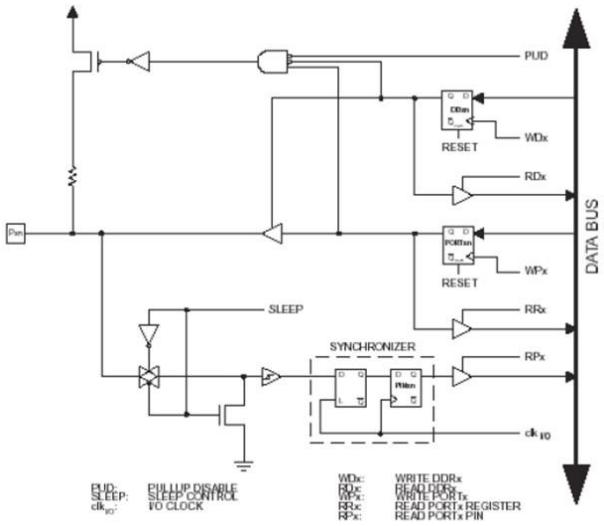
```
#include <avr/io.h>
```

Rejestry specjalne służą do konfigurowania i sterowania pracą peryferii mikrokontrolera (np. portów, liczników, przetwornika analogowo-cyfrowego, itd.). Jednym z podstawowych peryferii każdego mikrokontrolera są porty wejścia/wyjścia. Służą one między innymi do wysyłania i odbierania danych przez MCU. Mikrokontroler ATmega32 wyposażony jest w 4 porty (A, B, C, D). Do konfiguracji kierunku portu (wejście/wyjście) służy rejestr specjalny DDRx (Data Direction Register, gdzie x to nazwa portu np. dla portu A rejestr będzie nosił nazwę DDRA).



Rys. 3.1. Struktura logiczna portu mikrokontrolera

Jak wynika z funkcji portu do odczytu stanu linii we/wy (Pin.OUT) służy rejestr PINx. Rejestr ten odczytuje stan logiczny będący funkcją wyjścia rejestr PORTx oraz stanów logicznych na liniach we/wy uzależnionych od cyfrowych układów zewnętrznych dołączonych do tych linii.



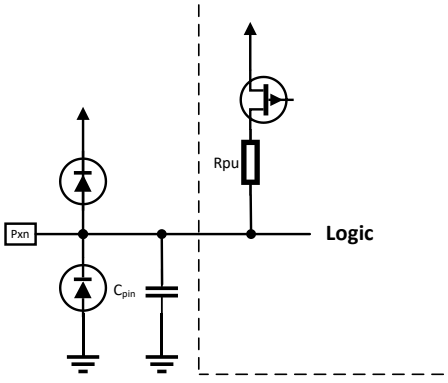
Rys. 3.2. Schemat ideowy portu mikrokontrolera

Tabela 2.1 Konfiguracja wyprowadzeń

DDRn	PORTn	PUD	I/O	pull-up	STAN
0	0	X	In	No	Tri - state
0	1	0	In	Yes	In - Low
0	1	1	In	No	Tri - state
1	0	X	Out	No	Out Low
1	1	X	Out	No	Out High

PUD – bit w rejestrze SFIOR. Umożliwia blokowanie funkcji pull-up dla wszystkich portów.

Każdemu z bitów w rejestrze DDRx odpowiada fizyczna nóżka (pin-out) wejścia/wyjścia mikrokontrolera, znajdująca się na porcie X. Np. bit nr 3 w rejestrze DDRC (czyli PC3) odpowiada nóżce podpisanej „Port C 3” na makiecie dydaktycznej.



Rys. 3.3. Struktura pin-outu portu mikrokontrolera

Końcówki są zabezpieczone przed przepięciami (diody ESD). Istnieje możliwość aktywacji rezystora podciągającego do VCC (pull-up).

Aby skonfigurować daną nóżkę jako wyjście należy wpisać w odpowiednie miejsce w rejestrze DDRC wartość logiczną '1'. Jeżeli wpisana zostanie wartość logiczna '0' to dana linia portu będzie skonfigurowana jako wejście. Do wpisywania wartości do rejestrów służy między innymi instrukcja:

```
DDRA=0xFF;
```

Użyta powyżej instrukcja powoduje przepisanie wartości do rejestru DDRA. Wykonanie tego fragmentu kodu spowoduje wysterowanie wszystkich 8 nóżek portu A (PA0..7) jako wyjścia.

Możliwe jest skonfigurowanie części nóżek portu jako wejścia, części jako wyjścia, np.:

```
DDRA=0x0f;   //konfiguracja wyprowadzeń portu A jako wejścia i
wyjścia
```

Taki zapis spowoduje, że 4 najstarsze nóżki portu A (linie PA7..5) będą wejściami, natomiast pozostałe będą wyjściami (linie PA4..0).

Jeżeli dana nóżka (nóżki, cały port) pracuje jako wejście stany logiczne odpowiadające sygnałom dostarczonym z zewnątrz do linii portu są wpisywane przez mikrokontroler do rejestru PINx. Aby odczytać wartość z portu i zapisać ją do rejestru uniwersalnego należy użyć instrukcji:

```
char x=PINA;
```

Jeżeli dana nóżka (nóżki, cały port) pracuje jako wyjście to wysyłane nią dane należy wpisywać do rejestru PORTx. W wypadku wysyłania danych np. portem A należy użyć rejestru PORTA (uwaga: port A" oznacza jedno z urządzeń wewnątrz mikrokontrolera, „PORTA" oznacza jeden z rejestrów sterujących portem A):

```
PORTA=0b01010101;
```

Wysłanie informacji 0b01010101 przy użyciu portu A.

```
DDRA=0x00;
PORTA=0xFF;
```

Instrukcja ta przy skonfigurowaniu portu jako wejście powoduje podciągnięcie do „1" wejścia portu. W stanie gdy do linii nie są podłączone zewnętrzne sygnały, odczytywany stan portu będzie wynosił: PINA=0xFF;

Wpisanie powyższego programu do mikrokontrolera i podłączenie jego portu A z wejściami sterującymi pracą diod LED – Złącze JP22 spowoduje, że zaświeci się co druga dioda (D1, D3, D5, D7), co druga nie będzie świeciła (D2, D4, D6, D8).
Podsumowując:

- mikrokontroler ATmega32 posiada 4 urządzenia typu 'PORT': portA, portB, portC, portD;
- do sterowania każdym z tych urządzeń służą 3 rejestry specjalne: DDRx, PORTx i PINx;
- bity w tych rejestrach oznaczamy w jednolity sposób: PC0, PB2, PA7, PD5, itp.

zakres 2, Operacje selektywne
W języku C jest sześć operatorów bitowych:|,&,^,<<,>>,~. Operatory te są szczególnie użyteczne przy manipulacjach bitami rejestrów. Poniżej na przykładach wyjaśnienie ich użycia. Oczywiście w przykładach liczby przedstawione są w postaci dwójkowej.

operator "|" - bitowa alternatywa (OR)

```
    0 1 0 1 0 1 0 1
|
    0 0 1 1 0 0 1 1
=
    0 1 1 1 0 1 1 1
```

operator "&" - bitowa koniunkcja (AND)

```
    0 1 0 1 0 1 0 1
&
    0 0 1 1 0 0 1 1
=
    0 0 0 1 0 0 0 1
```

operator "^" - bitowa alternatywa wykluczająca (XOR)

```
    0 1 0 1 0 1 0 1
^
    0 0 1 1 0 0 1 1
=
    0 1 1 0 0 1 1 0
```

operator "<<" - przesunięcie w lewo

```
 1 0 0 1 1 0 0 1  << 3 =  1 1 0 0 1 0 0 0
```

operator ">>" - przesunięcie w prawo

```
 1 0 0 1 1 0 0 1  >> 5 =  0 0 0 0 0 1 0 0
```

operator "~" - dopełnienie jedynkowe

```
 ~1 0 0 1 1 0 0 1  =  0 1 1 0 0 1 1 0
```

Operacje selektywne polegają na użyciu funkcji logicznych celem modyfikacji wskazanej w masce numerów bitów w rejestrach mikrokontrolera.

Operacja selektywnego ustawienia:

```
PORTD |= 0xF0;                          /* ustawia bity nr. 4..7 */
PORTD |= _BV(7)|_BV(6)|_BV(5)|_BV(4);
```

Operacja selektywnego zerowania:

```
PORTD &= 0xAA;                    /* zeruje bity nr. 0,2,4,6 */
PORTD &=BV(7)|_BV(5)|_BV(3)|_BV(1);
```

Operacja selektywnego negowania:

```
PORTD ^= 0x0F;                      /* neguje bity nr. 0..3 */
PORTD ^= _BV(3)|_BV(2)|_BV(1)|_BV(0);
```

zakres 3. Pętle licznikowe i instrukcje warunkowe.

W naszych prostych, przykładowych programach będzie można wyróżnić dwie sekcje, pierwsza to instrukcje wykonywane raz jeden, natychmiast po starcie programu, a druga sekcja to blok instrukcji wykonywany wielokrotnie w nieskończonej pętli. Nieskończoną pętle w programie można zbudować używając instrukcji "for" lub "while", w taki sposób, jak w przykładzie poniżej. W obu przypadkach instrukcje we wnętrzu pętli objęte są parą nawiasów klamrowych "{","}", podobnie jak zawartość całej funkcji głównej "main". Instrukcje tworzące pętle to: "for" i "while". Obie instrukcje pozwalają na utworzenie pętli nieskończonej:

```
int main(void)
{
  /* Instrukcje wykonywane raz jeden po starcie programu */

  /* Pętla nieskończona utworzona instrukcją "for" */
  for(;;)
  {
  /* Instrukcje w nieskończonej pętli */
  }
}

/* Pętla nieskończona utworzona instrukcją "while" */
  while(1)
  {
  /* Instrukcje w nieskończonej pętli */
  }
```

Do podejmowania decyzji (sterowania pracą programu) wykorzystywana jest instrukcja "if-else":

```
if( wartość_logiczna PRAWDA/FAŁSZ )

/* Instrukcja wykonywana jeśli PRAWDA */
else
/* Instrukcja wykonywana jeśli FAŁSZ */
```

Część "else" , czyli instrukcje wykonywaną gdy FAŁSZ można pominąć.

```
if( wartość_logiczna PRAWDA/FAŁSZ )
/* Instrukcja wykonywana jeśli PRAWDA */
```

Obejmując fragment kodu parą klamrowych nawiasów "{","}" tworzymy tzw. blok instrukcji, blok w "if-else" jest traktowany jako pojedyncza instrukcja.

```
if( wartość_logiczna PRAWDA/FAŁSZ )
{
/* Blok instrukcji wykonywany jeśli PRAWDA */
}
else
{
/* Blok instrukcji wykonywany jeśli FAŁSZ */
}
```

Chcąc sprawdzić czy jeden lub grupa bitów jednocześnie w rejestrze I/0 ma wartość "1" można to zrobić z użyciem instrukcji "if" w następujący sposób:

```
if(PINA & MASKA)
{
  /* Blok instrukcji wykonywany jeśli warunek spełniony */
}
```

Gdzie "PINA" to nazwa rejestru, a "MASKA" to stała wartość z ustawionymi tymi bitami, które potrzeba testować

Przykłady:

```
/* Jeśli bit nr. 3 rejestru PINA ma wartość "1" */
if(PINA & 0x08){}

/* Jeśli bity 0 lub 1 w PIND mają wartość "1" */
if(PIND & 0x03){}
```

```
unsigned char a, b;

/* Jeśli wartość zmiennej 'a' jest równa wartości zmiennej 'b',
w rejestrze PORTA zostanie zapisana wartość 0x01, w przeciwnym
razie w PORTB zostanie zapisana wartość 0x0F */

 if(a==b) PORTA = 0x01; else PORTB = 0x0F;
```

Pytania kontrolne:

1. Pytanie 1. Co to są operacje selektywne? Selektywnie ustaw bit 0 rejestru DDRA
2. Pytanie 2. W jaki sposób odczytujemy stan portu A. Jaka jest wymagana konfiguracja portu, gdy chcemy odczytać tylko linie 1 i 2.

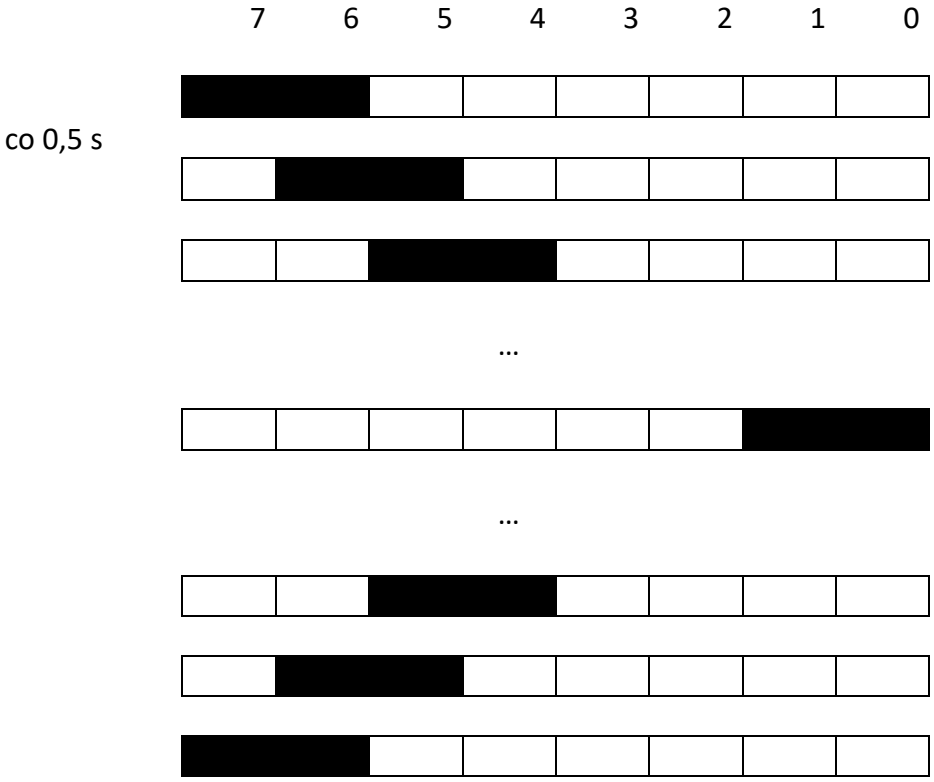
3. Pytanie 3. W jaki sposób dokonujemy testowania wybranych bitów w rejestrach specjalnych?

Zadanie 3.1. Programowanie portów mikrokontrolera. Operacje selektywne.

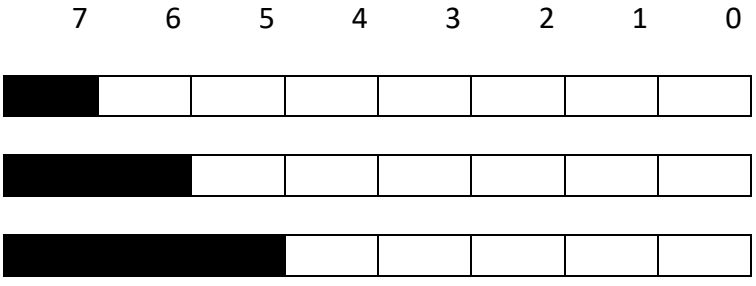
Treść zadania

Polecenie 1. Napisz, uruchom i przetestuj program sterujący liniami portu A, do których dołączono linijkę diodową D0…D7. Korzystając wyłącznie z operacji selektywnego: ustawienia, zerowania i negowania napisz kolejne programy w języku C dla poniższych przykładów od 1 do 5 w oparciu o rejestry specjalne Portu A. Do opóźnień czasowych wykorzystaj dostępne w bibliotece: delay.h funkcje opóźnień - _dely_ms(xxx);.
Każdy program należy wykonać w trybie krokowym obserwując (rejestrując) wybrane stany Portu A. W sprawozdaniu należy umieścić: kod źródłowy wraz z opisem oraz kod modułu main() w asm (należy skorzystać z deasemblera dostępnego w AvrStudio).

Przykład1

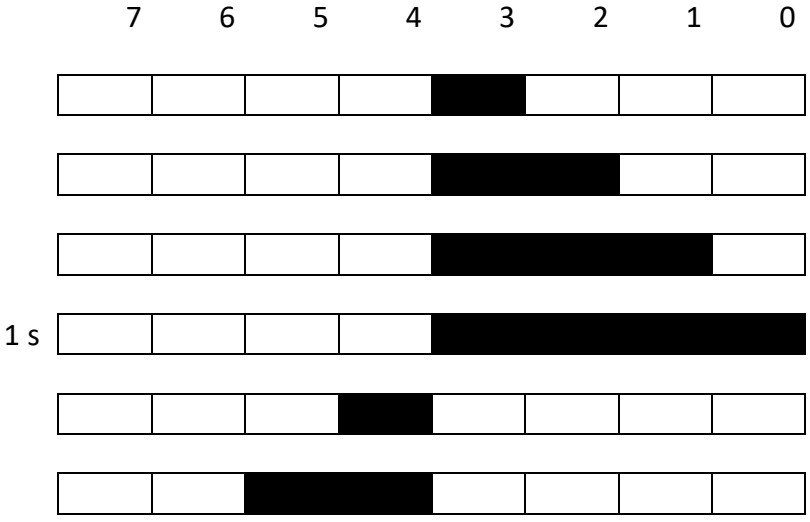


Przykład2

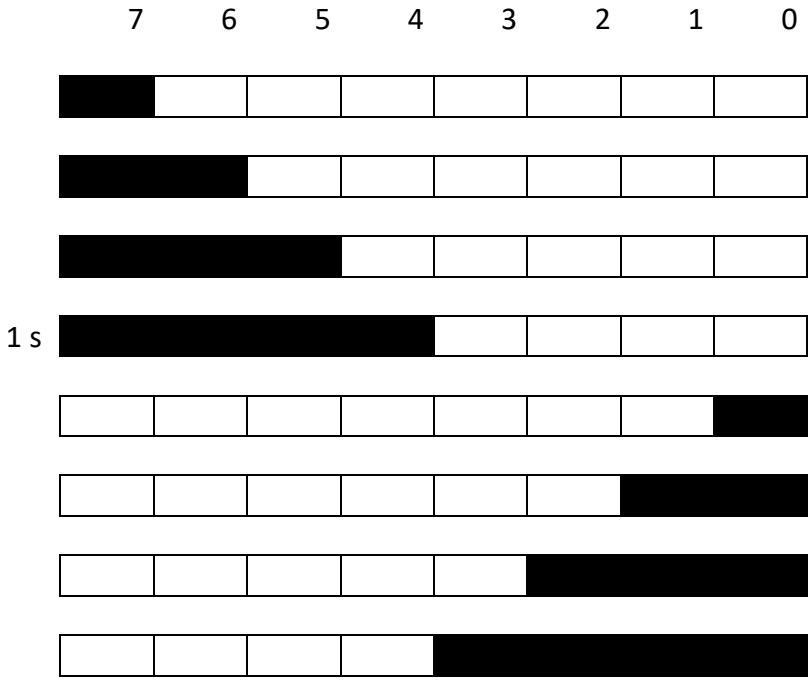


Przykład3

Przykład4



Przykład5



Polecenie 2. Na podstawie testów programu dokonaj rejestracji wybranych stanów portu A. Opracuj sprawozdanie na podstawie wskazówek prowadzącego laboratorium.

LABORATORIUM 4. UKŁADY PERYFERYJNE MIKROKONTROLERA AVR. TIMERY.

Cel laboratorium: Zapoznanie z programowaniem układów liczników/timerów mikrokontrolera ATmega32:

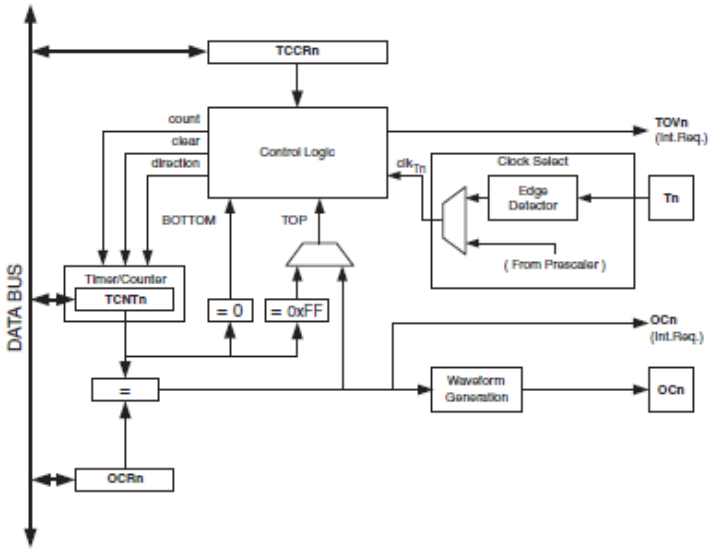
- **obsługa trybu pracy „normalny" wybranego licznika,**
- **obsługa trybu pracy „CTC" wybranego licznika,**
- **obsługa trybu pracy „PWM" wybranego licznika.**
- **generowanie sygnału o zadanej częstotliwości – konfiguracja licznika,**
- **generowanie sygnału o zadanej częstotliwości i współczynniku wypełnienia – konfiguracja licznika.**

Zakres tematyczny zajęć:

zakres 1, Budowa, zasada działania, konfiguracja liczników/timerów.

Podstawową funkcją liczników (Timer/Counter, T/Cn) jest zliczanie doprowadzonych impulsów. Jednak zazwyczaj liczniki pełnią jeszcze jedną funkcję w mikrokontrolerze (MCU, MicroController Unit) – generowanie przebiegów prostokątnych o zadanej częstotliwości lub o zadanym współczynniku wypełnienia. Mikrokontroler ATmega32 został wyposażony w 3 liczniki: T/C0, T/C1 oraz T/C2. Charakteryzują się one podobnymi funkcjami, podobną zasadą działania i są sterowane w niemal identyczny sposób.
Liczniki mogą być taktowane dwoma rodzajami sygnałów (asynchroniczny T/C2 – trzema). Są to impulsy taktujące mikrokontroler (ustawienia fabryczne dla ATmega32 to 1 MHz) lub impulsy doprowadzone z zewnątrz przez wejście Tn (np. T0). Są one doprowadzane do bloku ClockSelect gdzie przy pomocy multipleksera następuje ostateczny wybór rodzaju taktowania. Multiplekser sterowany jest przy pomocy bitów CSxx (zob. opis rejestru TCCR0).

Figure 27. 8-bit Timer/Counter Block Diagram



Rys. 4.1. Schemat blokowy licznika T0

Sygnał taktujący licznik (clk_Tn) dociera do układu sterującego urządzeniem (Control Logic). Każdy impuls zwiększa aktualny stan licznika (stan rejestru TCNTn) o 1. Jeżeli rejestr TCNTn osiągnie wartość maksymalną (np. 255 dla licznika 8-bitowego) i zostanie zwiększony jeszcze o 1 nastąpi jego przepełnienie – zostanie wyzerowany, ustawiona zostanie flaga przepełnienia TOVn. Jeżeli aktualny stan rejestru TCNTn zrówna się ze stanem rejestru OCRn zostanie ustawiona flaga informująca o zrównaniu (OCn).
W trybach pracy związanych z generowaniem sygnału prostokątnego zrównanie TCNTn oraz OCRn wykorzystuje się do formowania przebiegu na wyjściu OCn (pojawia się zbocze narastające lub opadające).
Podstawowe różnice między nimi przedstawia Tabela 4.1.

Tabela 4.1 Porównanie liczników/timerów mikrokontrolera ATmega 32

	T/C0	**T/C1**	**T/C2**
Zakres zliczania	8-bitowy	16-bitowy	8-bitowy
Tryby pracy	4 (normalny, CTC, 2xPWM)	15 (normalny, 2xCTC, 12xPWM)	4 (normalny, CTC, 2xPWM)
Praca Wejścia dla sygnału zewnętrznego (Tn)	Synchroniczna 1 (T0)	Synchroniczna 2 (T1, ICP1)	synchroniczna i asynchroniczna 0 (brak możliwości zliczanie impulsów zewnętrznych)
Wyjścia dla generowanego przebiegu prostokątnego	1 (OC0)	2 (OC1A, OC1B)	1 (OC2)
Dostępne dzielniki częstotliwości taktującej (prescaler)	5	5	7
Przerwanie od przepełnienia (TOVn)	1	1	1
Przerwania od zrównania stanu licznika z rejestrem OCRn (OCn)	1	2	1
Rejestry sterujące	TCCR0 TIMSK TCNT0 TIFR OCR0	TCCR1A TIMSK TCCT1B TIFR TCNT1 OCR1A OCR1B ICR1	TCCR2 TIMSK TCNT2 TIFR OCR2 ASSR2

Rejestry specjalne liczników/timerów T/C0 oraz T/C2.

Do sterowania licznikami wykorzystuje się rejestry. Rejestry poszczególnych liczników mają zbliżone nazwy (różnią się tylko ostatnimi znakami – oznaczającymi, do którego licznika należą), ponieważ w obrębie danego licznika pełnią identyczne funkcje. I tak:
• TCCRn – rejestry konfiguracyjne; tu określamy sposób pracy licznika itp; Dla liczników T/C0 oraz T/C2

TCCR0, TCCR2

7	6	5	4	3	2	1	0
FOCn	WGMn0	COMn1	COMn0	WGMn1	CSn2	CSn1	CSn0
W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- Oba liczniki mają po jednym rejestrze kontrolnym. Układ i funkcje bitów w **TCCR0** oraz **TCCR2** są identyczne.
  - FOCn – ręczne sterowanie stanem wyjścia OCn (działa tylko w trybach normalnych i CTC).
  - WGMn1:0 – wybór trybu pracy (normalny, CTC, Fast PWM, PhaseCorrect PWM.
  - COMn1:0 – wybór sposobu działania wyjścia OCn (zob. tab. w dokumentacji); dla generowania przebiegu prostokątnego w trybie CTC sugerowana opcja to „toggle" – zmiana stanu na przeciwny.
  - CSn2:0 – wybór sposobu taktowania licznika (zob. tab. w dokumentacji): jeden z kilku możliwych dzielników częstotliwości lub taktowanie sygnałem dochodzącym przez wejście T0 (tylko T/C0!)
- **TCNTn** – rejestry zliczające; czyli aktualny stan licznika; można zmodyfikować stan T/Cn wpisując tutaj żądaną wartość;
- **OCRn** – rejestry porównawcze; pod warunkiem uruchomienia odpowiedniego trybu pracy można w nich określić do jakiej wartości licznik ma zliczać; rejestry te biorą udział w formowaniu przebiegu prostokątnego generowanego na wyjściu OCn;
- **TIMSK** – rejestr maski przerwań liczników (odblokowywanie przerwań); wspólny dla wszystkich liczników; dokładny opis funkcji poszczególnych bitów znajduje się w dalszej części instrukcji;
- **TIFR** – flagi przerwań liczników (informacja o spełnieniu warunku przerwania); wspólny dla wszystkich liczników; dokładny opis funkcji poszczególnych bitów znajduje się w dalszej części instrukcji;
- **ICR1** – tylko T/C1; rejestr zatrzaskujący (zapamiętujący) aktualny stan licznika; zatrzaskiwanie stanu wywoływane sygnałem zewnętrznym;
- **ASSR2** – tylko T/C2; rejestr konfiguracyjny dla pracy asynchronicznej.

W rejestrach TCCRn, TIMSK oraz TIFR każdy bit służy do uruchomienia określonej funkcjonalności licznika. W rejestrach TCNTn, OCRn oraz ICR1 wszystkie bity służą do tego samego – przechowują wartość liczbową.

Rejestry zliczające TCNT (Timer/CouNter Register):

Dla każdego z trzech liczników mamy jeden rejestr, w którym odbywa się zliczanie:
- 8-bitowy TCNT0 dla T/C0,
- 8-bitowy TCNT2 dla T/C2.

Rejestry porównawcze OCR (Output Compare Register). Dla każdego z wyjść OCn liczników mamy po jednym rejestrze OCR. W umieszcza się wartość, do której licznik ma doliczyć.

Wykorzystywane m.in. do określania częstotliwości generowanego przebiegu prostokątnego (tryby CTC) lub współczynnika wypełnienia sygnału (tryby PWM):

- 8-bitowy OCR0 dla T/C0 (wyjście OC0),
- 8-bitowy OCR2 dla T/C2 (wyjście OC2).

Rejestr maski (aktywacji) przerwań TIMSK (TimerInterruptMaSK register):

7	6	5	4	3	2	1	0
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

TOIEn – aktywacja przerwania informującego o przepełnieniu rejestru zliczającego TCNTn.

OCIEn – aktywacja przerwania informującego o zrównaniu stanu rejestrów TCNTn i OCRn.

TICIE1 – aktywacja przerwania związanego z pinem IC1 i układem *Input Capture*.

Tabela 4.1 Bity ustawiające źródło sygnału taktującego i podział prescalera dla C/T0:

CS02	CS01	CS00	Opis
0	0	0	Licznik zatrzymany
0	0	1	Taktowanie CK
0	1	0	Taktowanie CK/8
0	1	1	Taktowanie CK/64
1	0	0	Taktowanie CK/256
1	0	1	Taktowanie CK/1024
1	1	0	Zewnętrzny sygnał T0 (opadające zbocze)
1	1	1	Zewnętrzny sygnał T0 (narastające zbocze)

Tabela4.2 Bity ustawiające źródło sygnału taktującego i podział prescalera dla C/T0:

CS22	CS21	CS20	Opis
0	0	0	Licznik zatrzymany
0	0	1	Taktowanie CK
0	1	0	Taktowanie CK/8
0	1	1	Taktowanie CK/32
1	0	0	Taktowanie CK/64
1	0	1	Taktowanie CK/128
1	1	0	Taktowanie CK/256
1	1	1	Taktowanie CK/1024

Tryb CTC (Clear Timer on Compare)

Licznik działa jak w trybie normalnym, z jedną różnicą – rejestr zliczający TCNTn jest zerowany za każdym razem, gdy jego stan zrówna się ze stanem rejestru OCRn. Powoduje to ograniczenie zakresu pracy licznika (np. dla 8-bitowego T/C0 już nie 0÷255, ale 0÷OCR0). Podstawowa zaleta to możliwość zliczania do zadanej wartości (OCR0). O jej osiągnięciu licznik informuje ustawiając flagę przerwania OCn w rejestrze TIFR.

Najważniejszą funkcją tego trybu pracy jest generowanie przebiegu prostokątnego na wyjściu OCn licznika. Jeżeli rejestr kontrolny TCCRn zostanie odpowiednio skonfigurowany to każde zrównanie stanu rejestrów TCNTn oraz OCRn będzie powodowało zmianę stanu na wyjściu OCn na przeciwny. Częstotliwość generowanego wówczas przebiegu prostokątnego będzie opisana wzorem przedstawionym w podrozdziale ClearTimer on CompareMatch (CTC) gdzie:

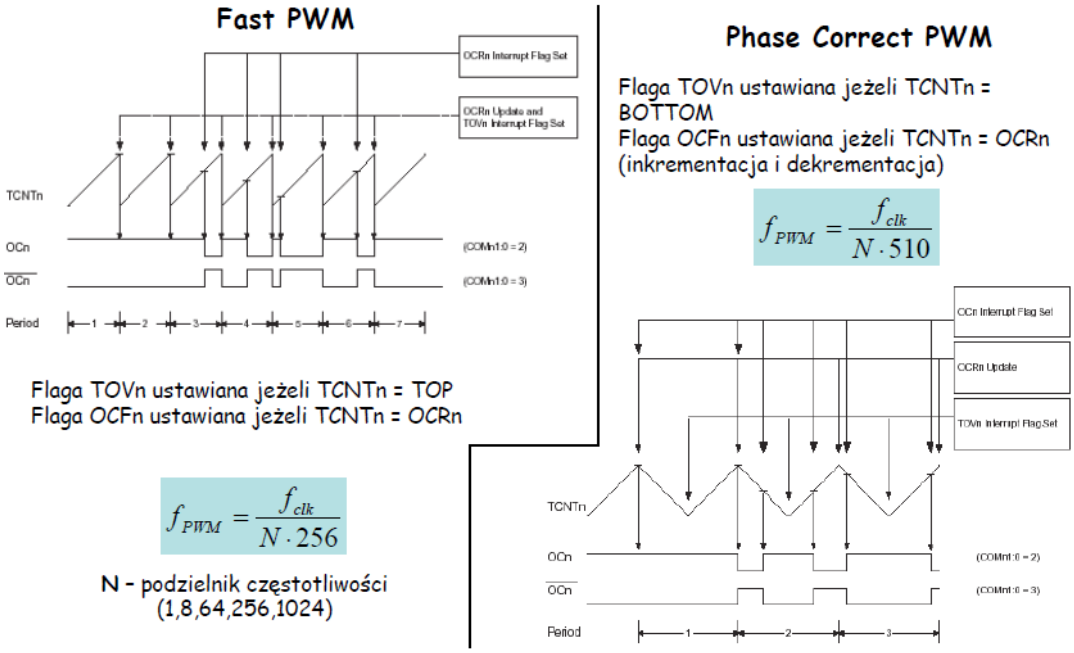
- fOCn – częstotliwość generowanego przebiegu;
- fclk_I/O – częstotliwość taktowania mikrokontrolera;
- N – wybrany dzielnik częstotliwości taktującej licznik (prescaler);
- OCRn – wartość wpisana do rejestru OCRn.

Uwaga: dla liczników T/C0 oraz T/C2 zakres pracy ustala się w odpowiednich rejestrach OCRn. Dla T/C1 służy do tego jeden z dwóch rejestrów (do wyboru) – OCR1A lub ICR1 (rejestr OCR1B nie może być użyty w trycie CTC).

Przykładowo, mikrokontroler taktowany częstotliwością 8 MHz, po ustawieniu prescalera na 1024 i wpisaniu do rejestru OCRn wartości „123” będzie formował na wyjściu OCn przebieg prostokątny o częstotliwości 31,5 Hz. Należy pamiętać o wysterowaniu OCn jako wyjście (odpowiedni rejestr DDRn).

Tryb PWM (Pulse Width Modulation)

Wyróżniamy 2 rodzaje trybów PWM – jednozboczowe (tylko zbocze narastające – np. tryb FAST PWM) oraz dwuzboczowe (narastające i opadające – np. tryb PC PWM). W kontekście laboratorium różnice między nimi są nieznaczne i ograniczają się do wzorów określających częstotliwość generowanego przebiegu. Tryby PWM służą do formowania na wyjściu OCn przebiegu prostokątnego o zadanej częstotliwości (jedynie zgrubnie) oraz o zadanym współczynniku wypełnienia (0-100%, dokładność nawet do 0,0015%). Sposób uruchomienia trybu oraz formowania sygnału na wyjściu zachodzi analogicznie jak w trybie CTC Również tutaj należy dokonać odpowiednich ustawień w rejestrach TCCRn oraz OCRn, należy pamiętać o wysterowaniu OCn jako wyjście.
Wykorzystując licznik T/C0 można ustalić jedynie 5 różnych częstotliwości dla F-PWM i 5 dla PC- PWM. Przy użyciu T/C2 – po 7 (w T/C0 i T/C2 decyduje o tym liczba możliwych ustawień prescaslera). Licznik T/C1 pozwala uzyskać znacznie lepszą dokładność, ponieważ częstotliwość ustalana jest nie tylko na podstawie prescalera, ale również rejestru określanego jako TOP (OCR1A, ICR1 lub wartość stała, w zależności od wybranego trybu pracy).



Rys. 4.2.Różnice pomiędzy trybami FAT PWM i Phace Correct PWM

Współczynnik wypełnienia dla sygnału PWM określa się dokonując wpisu do rejestru OCRn. Liczniki mogą pracować w zwykłym lub odwróconym trybie PWM. W przypadku trybu zwykłego wpisanie do OCRn wartości '0' powoduje formowanie sygnału o współczynniku wypełnienia 0%. Wpisanie wartości równej rejestrowi określonemu jako TOP – o współczynniku wypełnienia 100%. Pośrednie współczynniki wypełnienia można wyznaczyć z prostej proporcji. W przypadku trybu odwróconego sytuacja wygląda odwrotnie: OCRn = 0 => 100%, OCRn = TOP => 0%.

Przykładowo, mikrokontroler taktowany częstotliwością 4 MHz, po ustawieniu prescalera na 8, wpisaniu do rejestru OCRn wartości „50" i wybraniu trybu pracy „odwrócony" F-PWM licznika T/C2 będzie formował na wyjściu OCn przebieg prostokątny o częstotliwości 1953 Hz oraz współczynniku wypełnienia 80,4%. Należy pamiętać o wysterowaniu OCn jako wyjście (odpowiedni rejestr DDRn).

zakres 2, Omówienie przykładowych programów dla licznika/timera 0
Przykład1

```
#include <avr/io.h> //biblioteka zawierająca definicje i
rejestry procesora
#include <util/delay.h> //biblioteka pozwalająca na używanie
funkcji opóźnienia


void timer()
//procedura sprawdzająca przepełnienie flagi TOV0 i zerująca
zegar
{
     //sprawdzenie stanu flagi przepełnienia
     if(TIFR & (1 << TOV0))
```

```
     {   //jeśli flaga przepełnienia ustawiona, to:
     PORTA ^= _BV(0);            //zaświecenie pierwszej diody

     //ustawienie aktualnego stanu zliczonych impulsów licznika
w rejestrze TCNT0
     TCNT0 = 155;

     //wyzerowanie bitu flagi przepełnienia
     TIFR |= _BV(TOV0);
     }
}

int main(void)
{
/*ustawienie pinów sterujących diodami na zapis*/
     DDRA=0xFF;

/*ustaw.wart. na porcie A - wszystkie diody zgaszone*/
     PORTA=0x00;

//ustawienie PRESKALERA - 1024, co odpowiada 1ms
     TCCR0 |= (1 << CS02) | (1 << CS00);

//ustawienie wartości początkowej rejestru TCNT0 – 100 cykli -
end
     TCNT0 = 155;


while(1)                                  //pętla nieskończona
    { timer();                           //wywołanie funkcji timer()
     }
return 0;
}
```

Przykład 2.
Rozszerzenie przykłądu 1 o dodatkowy licznik cykli pracy timera.

```
#include <avr/io.h>
#include <util/delay.h>
#define F_CPU 1000000L

void odlicz(int il_sekund)
{
  TCCR0|=(1<<CS00)|(1<<CS02); //ustawienie preskalera
  int licz=0;
  TCNT0=155; //ustawienie wartości początkowej, licznik
odmierza do 255
  while(licz<=il_sekund*10)
     {
```

```
        if(TIFR&=(1<<TOV0)) //sprawdzenie czy ustawiona została
flaga przepełnienia
       {
                  TIFR|=(1<<TOV0);//zerowanie flagi
przepełnienia
                  TCNT0=155;// ponowne ustawienie wartości
początkowej
                   licz++;
                  }
                }
 PORTA^=0x01;
 TCCR0=0x00;
}

int main(void)
{
DDRA=0x01;
     PORTA=0x00;
while(1)
  {
  odlicz(5);//wywołanie procedury, w tym przypadku odmierza 5
sekund
  }
return 0;
     }
```

Przykład 3.
Celem kolejnego przykładu jest naprzemienne rozjaśnianie i przyciemnianie diody poprzez przypisywanie jej wartości odmierzonej przez Timer0 w trybie fast PWM. Stan miał się zmieniać od 0 do 255 co 4. Zmiana dokonywana jest co 10ms. Do odmierzenia czasu użyłem Timera2 w trybie Normal.

```
#include <avr/io.h>
#include <util/delay.h>
#define F_CPU 1000000L

//funckja czekania wykorzystująca Timer2

void czekaj(int il_ms)  {
 TCCR2|=(1<<CS00)|(1<<CS02);    //ustawieni preskalera
 int licz=0;
 TCNT2=245;                      //wartość poczatkowa

    while(licz<=il_ms*10)
{
if(TIFR&=(1<<TOV2))        //sprawdzanie flagi przepełnienia
  {
```

```
TIFR|=(1<<TOV2);          //zerowanie flagi przepełnienia
TCNT2=245;                   //ponowne ustawienie wartości
poczatkowej
licz++;
   }
 }
  TCCR2=0x00;//zerowanie timera po wykonanej operacji
odczekania
}

int main(void)
{
DDRB|=(1<<PB3);                 //ustawienie pin do podłączenia
diody
TCCR0|=(1<<WGM00)|(1<<WGM01);  //konfiguracja Timera0 w tryb
fastPWM
TCCR0|=(1<<COM00)|(1<<COM01);  //konfiguracja ustawienia OC0
TCCR0|=(1<<CS00);               //ustawnienie preskalera
int licz=0;

while(1)
{
while(licz<255)
    {
OCR0=licz;             //przypisanie wartości odliczanej do
OCR0 aby zmieniac stan diody
licz+=4;
czekaj(10);            // wykorzystanie procedury używającej
Timera2
}

while(licz>0)
{
OCR0=licz;     // przypisanie wartości odliczanej do OCR0 aby
zmieniac stan diody
licz-=4;
czekaj(10);          //wykorzystanie procedury używającej
Timera2
}
}
return 0;
}
```

Pytania kontrolne:

Pytanie 1. Na czym polega tryb NORMAL. Napisz konfigurację dla timera0 w tym trybie: ustawienie preskalera 1/8, pomiar 200 impulsów.

Pytanie 2. Jak przy pomocy licznika T/C0 w dowolnym trybie odliczyć 500 impulsów zegarowych;

Zadanie 4.1. Programowanie licznika/timera0 – bez systemu przerwań

Treść zadania

Polecenie 1.

- napisz procedurę timer10ms(); opóźnienia 10 ms dla timera 0 pracującego w trybie CTC, dla f=1MHz uruchom timer przy ustawieniu preskalera 1/256. Użyj procedury w pętli głównej programu do opóźnienia negowania linii PA0.
- korzystając z procedury timer10ms(); napisz procedurę czas_1s(); odmierzającej czas 1s poprzez kolejne zliczanie czasu 10 ms dokonywane przez procedurę mer10ms();

Polecenie 2

Licznik zdarzeń. Napisz program zliczający kolejne naciśnięcia przycisku Button 0, który należy dołączyć do linii T0 (linia PB0). Skonfiguruj timer 0 do pracy w trybie NORMAL i poprzez Preskaler przełącz źródło sygnałów wewn. na sygnały z linii zewnętrznej T0 (PB0). W pętli głównej programu co 200 ms (czas odmierz za pomocą _delaj_ms(200); odczytuj stan licznika TCNT0. Stan licznika poprzez port A wyświetl na linijce diodowej. Zliczania impulsów dokonaj/sprawdź dla zbocza narastającego oraz dla zbocza opadającego. Przy odczycie stanu licznika sprawdzaj stan flagi TOV0 (C/T0). Jeżeli wykryjesz stan flagi zatrzymaj timer.

Polecenie 3

Wygenerujmy kwadratowy kształt fali o okresie 200 ms i wypełnieniu 75% tzn. Czas trwania stanu 1 wynosi 150 ms a stanu 0 wynosi 50 ms. W tym celu:

- skorzystaj z trybu CTC,
- użyj preskalera: 1/1024,
- oblicz przy F_CPU = 1MHz wartości stanów liczników TCNT0 dla poszczególnych czasów.
- poprzez cykliczną zmianę rejestru OCR0 dla wyliczonych stanów 1 i 0 stwórz generator impulsów.

LABORATORIUM 5. SYSTEM PRZERWAŃ MIKROKONTROLERÓW AVR.

Cel laboratorium:

Celem laboratorium jest nabycie praktycznych umiejętności z programowania mikrokontrolerów z wykorzystaniem systemu przerwań.

Zakres tematyczny zajęć:

zakres 1, Zapoznanie studentów z systemem przerwań mikrokontrolera AVR

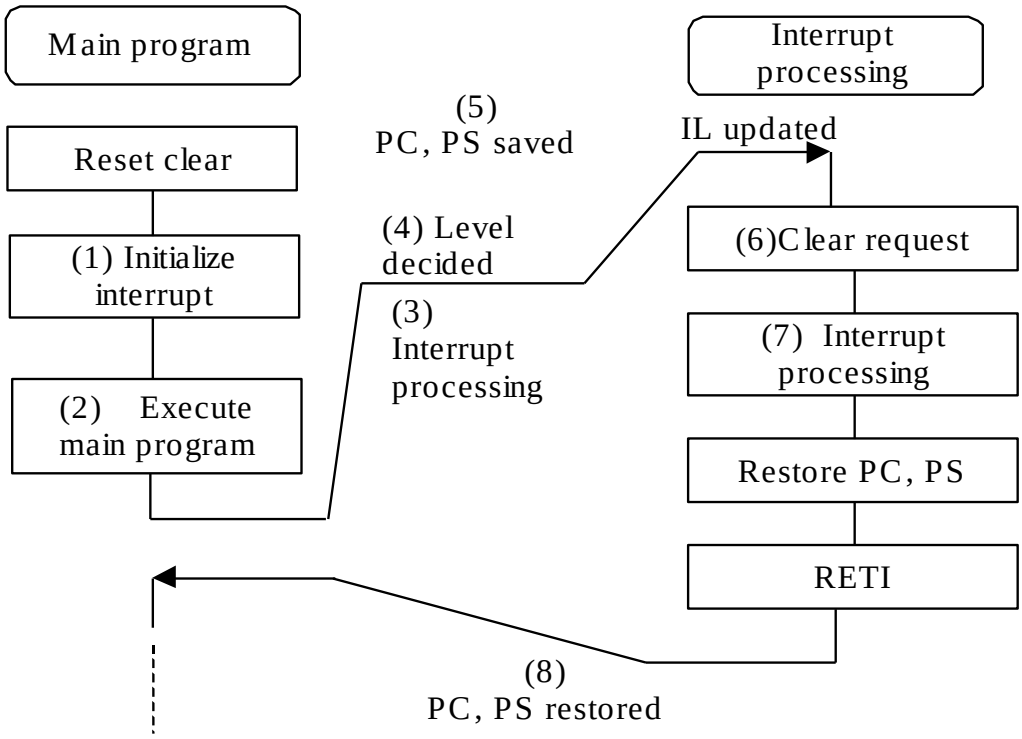
W trakcie pracy procesor wykonuje polecenia znajdujące się w głównej funkcji programu (main). Gdy pojawia się przerwanie (ang. interrupt), którym może być niemal dowolne zdarzenie, jak na przykład przepełnienie się licznika, zakończenie pracy przetwornika analogowo-cyfrowego czy też odczytanie wartości przez magistralę szeregową RS232, procesor wstrzymuje wykonywanie funkcji głównej i przechodzi do funkcji obsługi przerwania ISR (ang. Interrupt Service Routine). W ogólności funkcja ta wygląda w następujący sposób:

```
ISR (wektor_przerwania)
{

//definicja funkcji

}
```

Jest to rodzaj makra (pseudo-funkcji z parametrem). Wektor przerwania (czyli parametr funkcji ISR) definiuje nam jakie zdarzenie chcemy obsłużyć, czyli na jakie zdarzenie ma zareagować mikrokontroler. Wektor przerwania najprościej jest znaleźć w dokumentacji avr-libc. Gdy nastąpi zdarzenie odpowiadające danemu wektorowi przerwania, np. przepełnienie się licznika, mikrokontroler wykonuje kod znajdujący się w definicji funkcji ISR.

Funkcja obsługi przerwania powinna być jak najkrótsza (w sensie czasu wykonania, a nie długości kodu). Nie może zawierać **opóźnień!** Jeśli funkcja obsługi przerwania będzie zbyt długa to nie obsłużymy w porę przychodzących nowych przerwań. Natomiast pętla główna programu przestanie niemal być wykonywana. Mikrokontroler będzie jedynie wykonywał funkcję obsługi przerwania.

Ponadto w momencie pojawienia się przerwania obsługa przerwań jest domyślnie wyłączana. Czyli żadne nowe przerwanie nie zostanie obsłużone dopóki nie zakończy się aktualnie wykonywana funkcja obsługi przerwania. Co prawda możliwe jest programowe włączenie obsługi przerwania w przerwaniu (zagnieżdżanie przerwań) jednak jest to bardzo niewskazane gdyż rodzi wiele problemów!



Rys. 5.1. Działanie obsługi przerwań

Gdy podczas wykonywania obsługi jednego z przerwań przychodzi sygnał (wektor) innego przerwania, to z uwagi na wyłączone zagnieżdżanie przerwań jego obsługa zostanie wykonana dopiero po pewnym czasie. Najpierw dokończona zostanie obsługa pierwszego przerwania, następnie wykonana zostanie jedna instrukcja funkcji głównej (*main*) i dopiero wtedy nastąpi obsługa drugiego przerwania.

Jeżeli chcemy obsługiwać kilka różnych przerwań, na przykład dla Timera0 oraz Timera2 musimy zdefiniować dwie funkcje obsługi przerwania na przykład w następujący sposób:

```
ISR (TIMER0_COMP_vect)
{

//definicja funkcji obsługi Timera0

}
```

```
ISR (TIMER2_COMP_vect)
{

//definicja funkcji obsługi Timera2

}
```

Ponadto aby przerwania działały konieczne jest wywołanie funkcji sei(), która włącza globalną obsługę przerwań. Funkcję tę należy wywołać tylko raz w całym programie. Najlepiej na samym początku. W żadnym wypadku funkcja ta nie może znaleźć się w pętli while(1){ }. Ponadto obowiązuje prosta zasada: jedna procedura obsługi dla jednego źródła przerwań. Oznacza to, że dla Timera0, który posiada dwie flagi TOV0 i OCF0 możemy napisać dwie procedury odpowiednio dla: `TIMER0_OVF_vect` i `TIMER0_COMP_vect`.

Ważną rzeczą przy pisaniu funkcji obsługi przerwania jest świadomość, że procesor może przechowywać wartości zmiennych w rejestrach. Oznacza to, że jeżeli wewnątrz funkcji obsługi przerwania (ISR) zmienimy wartość jakiejś zmiennej globalnej to dla głównego programu (funkcji main()) wartość ta się nie zmieni gdyż procesor nadal będzie korzystał z zapisanej w rejestrze wartości.

(UWAGA: To czy zmienna będzie przechowywana w rejestrze procesora czy w pamięci, zależy od kompilatora oraz ustawień optymalizacji. Dla niektórych ustawień, np. braku optymalizacji, opisany poniżej efekt nie wystąpi, ponieważ dane zawsze będą pobierane z pamięci i zmienne nie będą zapisywane w rejestrach). Przeanalizujmy następujący kod:

```
#include <avr/io.h>
#include <avr/interrupt.h>

char temp;

ISR (TIMER0_OVF_vect)
{
temp++;        //wartość zmiennej temp jest imkrementowana
}

int main(void)
{

TIMSK|=1<<TOIE0;   //zezwolenie na obsługę przerwania od flagi
TOV0
TCNT0=155;         // wartość początkowa licznika Timera0

//start Timera0 przy podziale preskalera 1/1024
TCCR0|=_BV(CS00)|_BV(CS02);

sei();             // włączenie systemu przerwań do pracy

temp=20;

while (1)
{
//wewnątrz pętli zmienna "temp" ma wciąż wartość 20
}

return 0;
}
```

Pomimo, iż co jakiś czas wywoływana jest funkcja obsługi przerwania, która zwiększa wartość zmiennej temp, to wewnątrz funkcji main(), a dokładniej wewnątrz pętli while(1), wartość tej zmiennej wciąż jest równa 20, ponieważ procesor pobiera wartość z rejestrów, a nie z pamięci. Aby poradzić sobie z tym problemem możemy użyć polecenie volatile przed deklaracją zmiennej. Jest to instrukcja, która mówi kompilatorowi aby wartość zmiennej pobierał z pamięci, a nie z rejestrów. Deklaracja zmiennej temp powinna zatem wyglądać tak:

```
volatile uint8_t temp;
```

Kolejnym ważnym poleceniem jest static. Jeżeli chcemy korzystać ze zmiennej tylko wewnątrz funkcji obsługi przerwania, ale nie chcemy żeby za każdym razem była tworzona i inicjalizowana (definiowana), tylko aby wartość tej zmiennej przechowywana była pomiędzy wywołaniami funkcji obsługi przerwania, to możemy zadeklarować taką zmienną w następujący sposób:

```
ISR(wektor_przerwania)
{

static char temp=10;

temp++;

}
```

Zmienna temp przy pierwszym wywołaniu funkcji zostanie utworzona i zainicjalizowana (zdefiniowana) wartością 10, a następnie jej wartość zostanie zwiększona o 1. Przy kolejnych już wywołaniach funkcji wartość zmiennej temp będzie jedynie zwiększana o 1. Nie nastąpi ponowna definicja i podstawienie wartości 10.

zakres 2, System przerwań – AVR. Rejestry specjalne sterownika przerwań.

Cech systemu przerwań oraz wektory przerwań w mikrokontrolerze ATMEGA32, firmy Atmel.

- Zmiana stanu pewnych wejść (przerwania INT0, INT1, INT2, ICP1).
- Określony stan pewnych wejść (przerwania INT0, INT1).
- Przepełnienie licznika (TIMER0 OVF, TIMER1 OVF, TIMER2 OVF).
- Osiągnięcie przez licznik zadanej wartości (TIMER0 COMP, TIMER1 COMPA, TIMER1 COMPB, TIMER2 COMP).
- Zakończenie przetwarzania analogowo-cyfrowego (ADC).
- Zakończenie transmisji przez interfejs szeregowy (SPI STC, USART TXC, TWI).
- Odebranie danych przez interfejs szeregowy (USART RXC).
- Gotowość pamięci EEPROM (EERDY).
- Zmiana stanu wyjścia komparatora (ANACOMP).

Wektor przerwań mikrokontrolera ATmega32:

- W standardowej konfiguracji mikrokontrolera 42 najniższe adresy w pamięci programu zajmuje tablica przerwań.
- Im mniejszy adres, tym wyższy priorytet przerwania.
- Najwyższy priorytet ma RESET, potem przerwanie zewnętrzne INT0 itd.
- Przerwania są numerowane od 1 do 21.
- Obsługując przerwanie o numerze x, mikrokontroler ładuje do licznika programu wartość 2(x – 1)
- Powoduje to wykonanie rozkazu spod tego adresu w pamięci programu.
- Najczęściej jest to skok (rjmp lub jmp) do właściwej procedury obsługi przerwania.
- Uwaga: ATmega16 i ATmega32 obsługują ten sam zbiór przerwań, ale są one inaczej ponumerowane.

Table 18. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	$012	TIMER1 OVF	Timer/Counter1 Overflow
11	$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	$016	TIMER0 OVF	Timer/Counter0 Overflow
13	$018	SPI, STC	Serial Transfer Complete
14	$01A	USART, RXC	USART, Rx Complete
15	$01C	USART, UDRE	USART Data Register Empty
16	$01E	USART, TXC	USART, Tx Complete
17	$020	ADC	ADC Conversion Complete
18	$022	EE_RDY	EEPROM Ready
19	$024	ANA_COMP	Analog Comparator
20	$026	TWI	Two-wire Serial Interface
21	$028	SPM_RDY	Store Program Memory Ready

System przerwań – AVR: przerwania zewnętrzne.

Mikrokontroler ATmega32 posiada trzy źródła przerwań zewnętrznych - wyprowadzenia: INT0, INT1 INT2. Bity sterujące przerwaniami INT0 i INT1 znajdują się w rejestrze MCUCR:

Bit	7	6	5	4	3	2	1	0	
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
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	

Natomiast bit sterowania przerwaniem INT2 w rejestrze MCUCSR:

Bit	7	6	5	4	3	2	1	0	
	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF	MCUCSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	See Bit Description					

Przerwania zewnętrzne typu INTx mogą być wyzwalane określonymi sygnałami. Rodzaj sygnału zgłaszającego przerwanie programowany jest poprzez ustawienie bitów ISCn w rejestrach MCUCR i MCUCSR. Poniżej prezentuję sposób zgłaszania przerwania dla poszczególnych linii.

- dla linii INT0:

ISC01	ISC00	Sposób zgłaszania przerwania
0	0	Zgłaszanie niskim poziomem logicznym
0	1	Zgłaszanie negacją stanu logicznego
1	0	Zgłaszanie opadającym zboczem
1	1	Zgłaszanie narastającym zboczem

-dla linii INT1:

ISC11	ISC10	Sposób zgłaszania przerwania
0	0	Zgłaszanie niskim poziomem logicznym
0	1	Zgłaszanie negacją stanu logicznego
1	0	Zgłaszanie opadającym zboczem
1	1	Zgłaszanie narastającym zboczem

Sterowanie przerwaniami zewnętrznymi dokonujemy poprzez rejestr GICR:

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

W którym:

- INT1- bit maski przerwania INT1 (INT1=”1” i bit I=„1” przerwanie INT1 odmaskowane, INT1=„0”- zamaskowane).
- INT0- bit maski przerwania INT0 (INT0=”1” i bit I=„1” przerwanie INT0 odmaskowane, INT0=„0”- zamaskowane).
- INT2- bit maski przerwania INT2 (INT2=”1” i bit I=„1” przerwanie INT2 odmaskowane, INT2=„0”- zamaskowane).
-

Kolejnym ważnym rejestrem odpowiedzialnym za sterowanie przerwaniami zewnętrznymi jest GIFR:

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	INTF2	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

W którym:

- **INTF1:** bit zgłoszenia przerwania na wejściu INT1, ustawiany gdy przerwanie jest odmaskowane i zgłoszone, kasowany po wejściu do procedury obsługi lub poprzez zapis jedynki logicznej. Gdy przerwanie jest aktywne poziomem bit ten nie jest ustawiany.
- **INTF0:** bit zgłoszenia przerwania na wejściu INT0, ustawiany gdy przerwanie jest odmaskowane i zgłoszone, kasowany po wejściu do procedury obsługi lub poprzez zapis jedynki logicznej. Gdy przerwanie jest aktywne poziomem bit ten nie jest ustawiany.
- **INTF2:** bit zgłoszenia przerwania na wejściu INT2, ustawiany gdy przerwanie jest odmaskowane i zgłoszone, kasowany po wejściu do procedury obsługi lub poprzez zapis jedynki logicznej.

Mikrokontroler ATmega32 posiada wewnętrzne źródła przerwań sprzętowych. Są to źródła od flag wbudowanych układów peryferyjnych. Do najważniejszych ze wzg. na zakres ćwiczeń laboratoryjnych należą przerwania od timerów.

Rejestr maski przerwań od liczników czasomierzy to TIMSK:

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
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	

Poszczególne bity tego rejestrów odpowiadają flagom liczników/timerów sygnalizujących stan pracy tych układów, np.: flaga TOIE0 służy do udzielenia zezwolenia na obsługę przerwania od ustawienia flagi TOV0, kolejna flaga OCIE0 odpowiada bitowi OCF0 timera0.

Kolejnym rejestrem jest TIFR - rejestr znaczników przerwań od liczników-czasomierzy.

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
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	

Analiza przykładów obsług przerwań.

```
#include <avr/io.h>
#include <avr/interrupt.h>

}
ISR(INT0_vect) //podprogram obsługi INT0
{
PORTA^=0xAA; //zrób coś
}

ISR(TIMER0_OVF_vect)
{
```

```
PORTB^=0x55 ; //zrób coś innego
}


int main(void)
{
DDRD&= ~(1<<PD2);   //PD2 jako wejście
PORTD|= (1<<PD2);   //pull-up na PD2
MCUCR|= (1<<ISC00); //reakcja INT0 na dowolne zbocze
GICR|= (1<<INT0);   //odblokowanie przerwania INT0
TIMSK|= (1<<TOIE0); //odblokowanie przerwania od przepełnienia
flagi TOV0

sei();              //globalne odblokowanie przerwań

TCCR0|= (1<<CS00); //uruchomienie licznika T/C0

while(1)
{}
}
```

```
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(INT0_vect)
{

PORTA |= _BV(PA0); // zapal diodę LED

}

ISR(INT1_vect)
{

PORTA &= ~_BV(PA0); // zgaś diodę LED

}

int main() // program główny
{

DDRA |= _BV(DDRA0); // pina 0 portu A jako wyjście

GIMSK |= _BV(INT0)|_BV(INT1); // włączenie obsługi przerwań INT0
i INT1

MCUCR |= _BV(ISC11)|_BV(ISC10) | _BV(ISC01)|_BV(ISC00);
```

```
// włącz generowanie przerwań przez narastające zbocze na INT0
i INT1

sei(); // włącz globalną obsługę przerwań

while(1); // pętla nieskończona

return 0;

}
```

Pytania kontrolne:

1. Pytanie 1. Co to jest wektor przerwań? W jaki sposób piszemy procedurę przerwania od linii INT2?
2. Pytanie 2. W jaki sposób konfigurujemy system przerwań dla timera0 pracującego w trybie CTC?
3. Pytanie 3. Czym charakteryzuje się deklarowanie zmiennych globalnych stosowanych w procedurach przerwań i w module głównym main?

Zadanie 5.1 Programowanie przerwań mikrokontrolera ATmega32

Polecenie 1.
Przedstawiony kod dotyczy zadania, w którym program reaguje na zgłoszenia przerwań od linii INT0 i INT2. Do obu linii dołączono przyciski, które po naciśnięciu przez użytkownika powodują zmianę napięcia na wejściu danej linii, wymuszając reakcję programu poprzez wywołanie i obsługę procedury przerwania od danej linii. Proszę zwrócić uwagę, że pętla główna programu jest pusta. Moduł *main* zawiera jedynie nagłówek programu, w którym jest zawarta konfiguracja mikrokontrolera, w tym systemu przerwań. Wszelkie zadania są realizowane poprzez przerwania.

```
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(INT0_vect) {
PORTA ^= 1 << PA0;
}

ISR(INT2_vect) {
PORTA ^= 1 << PA2;
}

int main(void) {

uint8_t tmp;
 DDRA = 1 << PA0 | 1 << PA2;
tmp = MCUCR;
tmp &= ~(1 << ISC00);
```

```
    tmp |= 1 << ISC01;
    MCUCR = tmp;
    MCUCSR &= ~(1 << ISC2);
    GICR |= 1 << INT0 | 1 << INT2;
    GIFR |= 1 << INT0 | 1 << INT2;
    sei();

    while(1) {}
    }
```

zmodyfikuj podany przykład dla dwóch wariantów:
- przerwania IN0 i INT2 wyzwalane są zboczami narastającymi sygnału,
- przerwania IN0 i INT2 wyzwalane są zboczami opadającymi sygnału.

Wykorzystaj przyciski Buttons do zgłaszania przerwań.

Polecenie 2

W przerwaniu od Timera 0 pomierz czas:
- 100 ms w trybie CTC (jeden cykl)
- 1 s w trybie CTC (10 kolejnych cykli Timera)_.

Polecenie 3

W przerwaniu od Timera 0 odczytuj stan przycisków Buttons dołączonych do portu A i ich stan wyświetlaj w pętli głównej programu poprze port C. Przerwania należy generować co 5 ms.

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego