SPRAWOZDANIE

SYSTEMY WBUDOWANE

System przerwań mikrokontrolerów AVR.

IMIĘ I NAZWISKO: Jacek Wójcik

NUMER ĆWICZENIA: 7

Grupa laboratoryjna: 10

Data wykonania ćwiczenia: 24.11.2021

Spis treści

2. Zadanie 1 - kod

```c
#include <avr/io.h>             //Dołączam potrzebne biblioteki
#include <avr/interrupt.h>
ISR(INT0_vect) {                    // Przerwanie INT0
    PORTA ^= 1 << PA0;              // Zmieniam stan LED0
}
ISR(INT2_vect) {                    // Przerwanie INT2
    PORTA ^= 1 << PA2;              // Zmieniam stan LED2
}
int main(void) {
    DDRA = 1 << PA0 | 1 << PA2;     // Ustawiam pin 0 i pin 2 PORTA jako wyjście
    PORTB = 1 << PB2;
    PORTD = 1 << PD2;

    MCUCR |= 1 << ISC01;            // Ustawiam przerwanie INT0 na zbocze opadające
    MCUCSR &= ~(1 << ISC2);         // Ustawiam przerwanie INT2 na zbocze opadające
    //MCUCR |= 1 << ISC01 | 1 << ISC00; // Ustawiam przerwanie INT0 na zbocze narastające
    //MCUCSR |= 1 << ISC2;              // Ustawiam przerwanie INT2 na zbocze narastające


    GICR |= 1 << INT0 | 1 << INT2;  // Włączam przerwanie INT0 i INT2
    GIFR |= 1 << INT0 | 1 << INT2;  // Wywołanie przerwania INT0 i INT2 kiedy to będzie możliwe
    sei();                          // Właczenie obsługi przerwań
    while(1) {
        asm volatile("nop");        // Pętla główna
    }
}

```

3. Zadanie 1 - opis

Zadanie 1 polegało na zmodyfikowaniu programu z skryptu dla dwóch wariantów:

- przerwanie INT0 i INT2 jest wywoływane zboczem narastającym sygnału
- przerwanie INT0 i INT2 jest wywoływane zboczem opadającym sygnału

W zadaniu zaimplementowałem wariant z zboczem opadającym, a wariant z zboczem narastającym postanowiłem zakomentować.

Do programu ze skryptu wprowadziłem następujące zmiany:

1. W linijce 11 i 12 programowo podciągnąłem przyciski odpowiadające za przerwanie INT0 i INT2 pod zasilanie.

```
11      PORTB = 1 << PB2;               // Podciągam piny PB2 i PD2 programowo pod zasilanie
12      PORTD = 1 << PD2;
```

2. W linijce 14 i 15 ustawiłem odpowiednio przerwanie INT0 i INT2 na zbocze opadające poprzez zmodyfikowanie rejestru MCUCR dla INT0 i MCUCRS dla INT2.

```
14      MCUCR |= 1 << ISC01;            // Ustawiam przerwanie INT0 na zbocze opadające
15      MCUCSR &= ~(1 << ISC2);         // Ustawiam przerwanie INT2 na zbocze opadające
```

3. Tak samo w linijce 16 i 17 ustawiłem przerwania INT0 i INT2 na zbocze narastające, jednak jako że to nadpisze poprzednie ustawienie, zakomentowałem te dwie instrukcje.

```
16      //MCUCR |= 1 << ISC01 | 1 << ISC00; // Ustawiam przerwanie INT0 na zbocze narastające
17      //MCUCSR |= 1 << ISC2;              // Ustawiam przerwanie INT2 na zbocze narastające
```

4. W linijce 24 dodałem instrukcję w asemblerze, która powoduje, że w każdej iteracji pętli while procesor nic nie wykonuje przez jeden cykl. Dzięki temu możliwe jest debugowanie programu w różnych symulatorach, ponieważ w innym przypadku debugger po wejściu w pętlę while nie będzie miał instrukcji, na której może się zatrzymać.

```
23      while(1) {
24          asm volatile("nop");        // Pętla główna
25      }
```

3. Zadanie 2 - kod

```c
#define F_CPU 1000000       // Ustawiam częstotliwość na tę z labolatorium
#include <avr/io.h>     // Dołączam potrzebne biblioteki
#include <avr/interrupt.h>

volatile char number = 1;   // Inicjalizuję licznik cykli timera

ISR (TIMER0_COMP_vect) {    // Przerwanie TIMER0 co 100ms
    PORTA ^= 0x01;          // Migam LED0 co 100 milisekund
    if(number == 10)            // Odliczam 10 kolejnych cykli timera
    {
        PORTA ^= 0x02;      // Migam LED1 co 1 sekundę
        number = 1;
    }
    else
    {
        number++;               // Odliczam kolejny cykl
    }
}

int main(void)
{
    DDRA = 0x03;    // Ustawiam pin 0 i 1 PORTA jako wyjscie

    OCR0 = 98;      // ustawienie momentu przerwania na 98 (około 0,1s)

    TIMSK |= 0x02;  // włączenie przerwania TIMER0

    //  Włączam timer 0 w trybie CTC z preskalerem 1/1024
    TCCR0 |= 1 << WGM01 | 1 << CS02 | 1 << CS00;

    sei();                      // włączam obsługę przerwań


    while (1)
    {
        asm volatile("nop");
    }
}

```

4. Zadanie 2 - opis

Zadanie 2 polegało odmierzeniu za pomocą timera0 dwóch okresów czasu:

- 100ms jako jeden cykl timera w trybie CTC
- 1s jako 10 cykli timera w trybie CTC

Założyłem, że w celu pokazania działania programu będę naprzemiennie zalapalał i gasił LED-y w następującym porządku:

- LED0 sterowany przez pin 0 PORTA, zmiana stanu co 100ms
- LED1 sterowany przez pin 1 PORTA, zmiana stanu co 1s

W tym celu na początku programu oznaczyłem oba piny jako wyjście. Następnie obliczyłem wartość rejestru OCR0 z wzoru wspomnianego we wcześniejszym sprawozdaniu, tj.

```
OCR0 = <czas> * <częstotliwość mikrokontrolera> * <preskaler>
```

I na podstawie tego dla czasu 100ms będzie to wartość:

```
OCR0 = 0,1s * 1000000Hz * 1/1024 = 97,66 ≈ 98
```

Jako że OCR0 przyjmuje jedynie wartości całkowite, musiałem zaokrąglić wynik do 98, co daje mi faktyczny czas 100,3ms.

Na koniec inicjalizacji włączyłem obsługę przerwań funkcją **sei()**.

Następną częścią było implementacja zawartości przerwania. Zmiana stanu LED0 polegała na wybiórczej negacji pierwszego bitu PORTA w każdym cyklu timera. Zmiana stanu LED1 wymagała zliczania ilości cykli w przerwaniu. W celu wykonania tego w bezpieczny sposób, oznaczyłem zmienną licznika jako wartość **volatile**.

Znaczenie tego wyjaśniłem w poprzednim sprawozdaniu, jednak przypominając: jeżeli zmienna jest oznaczona jako **volatile**, nie podlega optymalizacjom kompilatora, dzięki czemu nie ma ryzyka że zmiana jej wartości podczas przerwania zostanie zapomniana, ponieważ kompilator np. przechował wartość tej zmiennej w rejestrze procesora pomiędzy dwoma instrukcjami, pomiędzy którymi wykonało się przerwanie i przez to że wartość tej zmiennej w następnych instrukcjach po zakończeniu przerwania została odczytana z rejestru, a nie z pamięci RAM, to zmiana tej wartości w przerwaniu przepadła.

5. Zadanie 3 - kod

```c
#define F_CPU 1000000       // Ustawiam częstotliwość na tę w labolatorium
#include <avr/io.h>         // Dołączam potrzebne biblioteki
#include <avr/interrupt.h>

volatile char buttons = 0;  // Tworzę zmienną do zapamiętania stanu PINA

ISR (TIMER0_COMP_vect) {    // Przerwanie TIMER0 co 5ms
    buttons = ~PINA; // Zapamiętuję zanegowany stan PIN
}

int main(void)
{
    DDRC |= 0x03;       // Ustawiam pin 0 i 1 PORTA jako wyjscie

    PORTA |= 0x03;      // Podciągam przyciski pod zasilanie

    OCR0 = 5;           // ustawienie momentu przerwania na 5 (około 5ms)

    TIMSK |= 0x02;      // włączenie przerwania TIMER0

    //  Włączam timer 0 w trybie CTC z preskalerem 1/1024
    TCCR0 |= 1 << WGM01 | 1 << CS02 | 1 << CS00;

    sei();              // włączam obsługę przerwań

    while (1)
    {
        // Zmieniam stan tylko dwóch podłączonych LEDów w PORTC
        PORTC = (PORTC & 0xFC) | (buttons & 0x03);
    }
}

```

6. Zadanie 3 - opis

Zadanie 3 polegało na odczytywaniu stanu przycisków dołączonych do portu A i wyświetlanie ich stanu w pętli głównej programu przez port C. Odczytywanie stanu należało wykonać w przerwaniu od timera 0, które ma być wyzwalane co 5ms.

Jako że w zadaniu nie została sprecyzowana liczba przycisków, która ma zostać podłączona do PORTA, zdecydowałem się na użycie 2 przycisków, podłączonych do pinów 0 i 1 PINA, wyświetlając ich stan na LED0 i LED1, które są podpięte pod piny 0 i 1 PORTC.

Zadanie rozpocząłem od zainicjalizowania portów, tzn. ustawiłem wyżej wymienione piny PORTC jako wyjście i podciągnąłem wyżej wymienione piny PORTA pod zasilanie.

Następnie zainicjalizowałem timer0 w trybie CTC z preskalerem 1/1024. Wartość, jaką należy wpisać do OCR0 w celu odczekania 5ms obliczyłem jak w zadaniu 2, tzn. z wzoru:

`OCR0 = <czas> * <częstotliwość mikrokontrolera> * <preskaler>`

I na podstawie tego dla czasu 5ms będzie to wartość:

`OCR0 = 0,005s * 1000000Hz * 1/1024 = 4,88 ≈ 5`

Jako że OCR0 przyjmuje jedynie wartości całkowite, musiałem zaokrąglić wynik do 5, co daje mi faktyczny czas 5,12ms.

Na koniec inicjalizacji włączyłem obsługę przerwań funkcją **sei()**.

W funkcji obsługującej przerwanie zapamiętałem wartość PINA w zmiennej **buttons**. Ważne są tutaj dwie rzeczy:

- Po pierwsze, tak samo jak w zadaniu 2 musiałem oznaczyć zmienną **buttons** jako **volatile** w celu uniknięcia optymalizacji kompilatora.
- Po drugie, jako że przyciski są programowo podciągnięte do zasilania, stan na PINA to domyślnie 1. Gdybym przeniósł ten stan na PORTC, skończyłoby się to sytuacją, gdzie LED-y świecą, gdy przycisk nie jest wciśnięty, a gaszą się, gdy wciskam przycisk. Żeby uniknąć tej sytuacji, zapamiętuję w zmiennej **buttons** zanegowaną wartość PINA.

W pętli głównej przypisuję do PORTC wartość zmiennej **buttons** tak, żeby nie nadpisać pozostałej części portu.

7. Wnioski

Podczas tych laboratoriów nauczyłem się następujących rzeczy:

- Podczas wykonywania się jednego przerwania jest możliwość wywołania się innego przerwania (np. mamy dwa przyciski które można naciskać niezależnie).

  Zazwyczaj jest to niepożądany efekt, ponieważ z powodu np. aktualnego stanu portów, jaki tymczasowo jest ustawiany podczas przerwania, takie zagnieżdżone przerwanie może zostać niepotrzebnie wywołane i zakłócić obsługę faktycznego przerwania.

  Żeby temu zapobiec, należy na samym początku obsługi przerwania wywołać funkcję **cli()**, która zablokuje obsługę innych przerwań, a następnie na koniec obsługi przerwania wywołać funkcję **sei()**, która umożliwi obsługę innych przerwań.

  Podczas zadań, które wykonywałem na laboratorium, wyłączanie obsługi przerwań podczas obsługi przerwania nie było konieczne.
- Nauczyłem się również, by obsługę przerwań włączać dopiero po zainicjalizowaniu programu, ponieważ w innym razie może dojść do błędnego działania programu z powodu wywołania się przerwania przed końcem inicjalizacji (np. wyświetlacz LCD nie zostanie zainicjalizowany a w obsłudze przerwania będzie konieczność użycia wyświetlacza).
- W związku z koniecznością odczytu i zapisu zmiennych podczas obsługi przerwań, musiałem w moim programie oznaczać takie zmienne modyfikatorem **volatile**, dzięki czemu miałem pewność, że modyfikacje do zmiennej wprowadzane w przerwaniu nie zostaną pominięte w głównym przebiegu programu. Więcej o tym mechanizmie napisałem we wnioskach z laboratorium 5 i skróciłem to w opisie zadania 2 z tego laboratorium.
- Dowiedziałem się, że odpowiednio modyfikując stan rejestru GIFR mogę wywoływać obsługę przerwania. Jednocześnie mogę samodzielnie obsługiwać przerwania nawet z wyłączoną obsługą przerwań, wystarczy że będę sprawdzał czy w rejestrze GIFR są ustawione odpowiednie flagi.

  Dzięki temu będę mógł np. dostosować obsługę przerwania INT0, podczas którego wyłączam obsługę przerwań, ale za pomocą GIFR mogę zobaczyć, że jednocześnie został spełniony warunek wywołania przerwania INT1 i dostosować obsługę przerwania INT0.
- Przypomniałem sobie, w jaki sposób za pomocą maskowania zapisać informacje tylko do części bitów danego portu, nie naruszając pozostałych informacji zapisanych na reszcie bitów.
- Nauczyłem się, w jaki sposób, wykorzystując własny licznik, mogę wykorzystać dany timer do jednoczesnego odliczania kilku różnych okresów czasu, wystarczy że są to wielokrotności najmniejszego odliczanego okresu czasu. (np. w zadaniu 2 było to 100ms odliczane w przerwaniu i 1s odliczana jako 10 przerwań 100ms).