SPRAWOZDANIE

SYSTEMY WBUDOWANE

Obsługa układów peryferyjnych.
Klawiatura.

IMIĘ I NAZWISKO: Jacek Wójcik

NUMER ĆWICZENIA: 5

Grupa laboratoryjna: 10

Data wykonania ćwiczenia: 10.11.2021

Spis treści

1. Zadanie 1 - kod

1.1 main.c

```
#include "lcd.h"                    // Dołączam moją bibliotekę
#include <avr/interrupt.h>          // biblioteka do przerwań
#include <stdlib.h>                 // bliblioteka do opóźnień

volatile char number  = 0;      // zapamiętuję ostatnio wciśnięty klawisz

void show_number(char num)      // funkcja wyświetlająca numer
{
    if(num != number)           // jeżeli obecny kod klawisza jest inny od poprzedniego
    {
        LCD_CLEAR();                // czyszczę wyświetlacz
        char buffer[3];             // tworzę bufor dla funkcji itoa
        itoa(num, buffer, 10);      // przetwarzam kod klawisza na ciąg znaków
        buffer[2]= '\0';            // dodaję pusty znak na koniec żeby utworzyć poprawny ciąg znaków w C
        LCD_WRITE(buffer);          // wyświetlam znak
    }
    number = num;
}

int debinary(int x)         // wyodrębniam numer bitu na podstawie wartości dziesiętnej połowy bajta
{
    if(x == 8) {            // 4 bit
        return 4;
    }
    if(x == 4) {            // 3 bit
        return 3;
    }
    return x;               // 2 lub 1 bit
}

ISR (TIMER0_COMP_vect) {                        // Przerwanie TIMER0

    if((PIND & 0x0F) != 0x0F)                       // Wykrywam że na jedenym z wierszy jest logiczne 0
    {
        char row = debinary(~PIND & 0x0F);                  // Zapamiętuję który to wiersz
        char backup_DDRD = DDRD;                            // Zapisuję stany portów
        char backup_PORTD = PORTD;

        DDRD ^= 0xFF;                           // Zamieniam linie wierszy
        PORTD ^= 0xFF;                          // Zamieniam podciągnięcia linii wierszy

        char column = debinary((PIND & 0xF0) >> 4);             // Odczytuję stan linii kolumn
        show_number((row-1)*4 + column);                    // Wyświetlam kod znaku na wyświetlaczu
        DDRD = backup_DDRD;                         // Przywracam zapisane stany linii
        PORTD = backup_PORTD;
    }
}
```

```
48
49  int main(void)
50  {
51      DDRD = 0xF0;                // Ustawiam starsze bity linii D jako wyjście
52
53      PORTD |= 0x0F;              // podciągam młodsze bity PORTD pod zasilanie
54
55      OCR0 = 195;                 // ustawienie momentu przerwania na 195 (0,05s)
56
57      TIMSK |= 0x02;              // włączenie przerwania TIMER0
58
59      TCCR0 |= 0b01000101;        // włączam tryb CTC i ustawiam prescaler na 1/1024
60
61      sei();                      // włączam obsługę przerwań
62
63      LCD_INIT();                 // Inicjalizuję wyświetlacz
64      while (1)
65      {
66          asm volatile("nop");    // Wykonuję niekończoną pętlę przez resztę czasu
67      }
68  }
69
70
```

1.2 lcd.h

```
1   #ifndef LCD_H_
2   #define LCD_H_
3
4   #define F_CPU 1000000           // Ustawiam częstotliwość na zgodną z zestawem w laboratorium
5
6   #include <avr/io.h>             // Dołączam biblioteki do obsługi portów i opóźnień
7   #include <util/delay.h>
8
9   void LCD_CLEAR();               // Funkcja czyszcząca cały wyświetlacz
10
11  void LCD_INIT();                // Funkcja inicjalizująca wyświetlacz
12
13  void LCD_WRITE(char *str);      // Funkcja wypisująca ciąg znaków na wyświetlacz
14
15  void LCD_XY(int x, int y);      // Funkcja ustawiająca kursor na danej pozycji na wyświetlaczu
16
17  void LCD_CLEAR_XY(int x, int y);    // Funkcja czyszcząca wyświetlacz od danej pozycji
18
19  #endif
20
```

1.3 lcd.c

```c
#include "lcd.h"

void sendHalfByte(char data)
{
    PORTA |= 0x02;                              // Wysyłam flagę ENABLE
    PORTA = (PORTA & 0x0F) | (data & 0xF0);     // Wysyłam pół bajta
    PORTA &= 0xFD;                              // Usuwam flagę ENABLE
}

void sendByte(char data)
{
    sendHalfByte(data);                         // Wysyłam górną połowę bajtu
    _delay_ms(2);                               // Opóźnienie w celu wykrycia stanu "0" na pinie ENABLE
    sendHalfByte(data << 4);                    // Wysyłam dolną połowę bajtu
}

void sendCommand(char data)
{
    sendByte(data);                             // Wysyłam komendę
    _delay_ms(5);                               // Czekam na przetworzenie komendy
}

void sendChar(char data)
{
    sendByte(data);                             // Wysyłam znak
    _delay_ms(2);                               // Czekam na wypisanie znaku
}

void LCD_CLEAR()
{
    PORTA &= 0xFE;                              // Przełączam LCD w tryb wprowadzania komend
    _delay_ms(2);                               // Opóźnienie w celu wykrycia stanu "0" na pinie RS
    sendCommand(0x01);                          // Czyszczę wyświetlacz
    PORTA |= 0x01;                              // Przełączam LCD w tryb wprowadzania danych
}


void LCD_INIT()
{
    _delay_ms(15);                          // Czekam aż wyświetlacz LCD zostanie zainicjalizowany

    DDRA |= 0xF3;                           // Ustawiam część linii A

    // Inicjalizacja standardowymi bajtami
    sendHalfByte(0x30);
    _delay_ms(5);                           // Czekam zgodnie z dokumentacją
    sendHalfByte(0x30);
    _delay_ms(1);
    sendCommand(0x32);

    // Inicjalizacja ustawień wyświetlacza
    sendCommand(0x28);                      // Ustawiam tryb 2 linii i znaków 5x8
    sendCommand(0x08);                      // Wyłączam wyświetlacz zgodnie z dokumentacją
    sendCommand(0x01);                      // Czyszczę wyświetlacz
    sendCommand(0x06);                      // Ustawiam kierunek wyprowadzania tekstu i sposób wyprowadzania na wyświetlacz
    sendCommand(0x0C);                      // Włączam wyświetlacz i ustawiam znak na pozycji kursora na miganie
    PORTA |= 0x01;                          // Przełączam LCD w tryb wprowadzania danych
}

void LCD_WRITE(char *str)
{
    while(*str)                             // Dopóki nie dojdę to null terminatora wypisuję kolejne znaki
    {
        sendChar(*str++);                   // Przesuwam się do kolejnego znaku po wypisaniu go na wyświetlaczu
    }
}

```

```
67  void LCD_XY(int x, int y)
68  {
69      PORTA &= 0xFE;                          // Przełączam LCD w tryb wprowadzania komend
70      _delay_ms(2);                           // Opóźnienie w celu wykrycia stanu "0" na pinie RS
71      sendCommand(0x80 | x | y << 6);         // Ustawiam kursor na odpowowiedniej pozycji
72      PORTA |= 0x01;                          // Przełączam LCD w tryb wprowadzania danych
73  }
74
75  void LCD_CLEAR_XY(int x, int y)
76  {
77      LCD_XY(x,y);                            // Ustawiam kursor na pozycji od której chcę czyścić
78      if(y == 0)                              // Sprawdzam czy to pierwsza linia
79      {
80          while(x++ < 16)                     // Dopóki nie dojdę do końca linii
81          {
82              LCD_WRITE(" ");                 // Wypisuję pusty znak na wyjście wyświetlacza
83          }
84          x = 0;                              // Ustawiam pierwszą składową adresu na początek linii
85          y++;                                // Ustawiam drugą składową adresu na drugą linię
86          LCD_XY(x,y);                        // Ustawiam kursor na podany adres -> początek drugiej linii
87      }
88      while(x++ < 16)                         // Dopóki nie dojdę do końca linii
89      {
90          LCD_WRITE(" ");                     // Wypisuję pusty znak na wyjście wyświetlacza
91      }
92  }
93
94
```

2. Zadanie 1 - opis

Zadanie 1 polegało na wyświetlaniu na ekranie LCD kodu znaku, który został wciśnięty na klawiaturze.
W celu wykrycia naciśnięcia klawisza stosuję algorytm odczytu klawiatury z odwracaniem kierunków linii w porcie. Polega on na podłączeniu wierszy klawiatury jako wejście do portu (ja stosuję PORTD) podciągnięte do zasilania, a kolumn klawiatury jako wyjście portu o stanie zero. Wtedy żeby sprawdzić, czy został naciśnięty jakikolwiek przycisk wystarczy sprawdzić czy którykolwiek wiersz został zwarty do stanu 0, czyli do jakiejkolwiek kolumny. Jeżeli tak jest, można sprawdzić który to dokładnie wiersz poprzez iloczyn zanegowanego stanu pinów i maski 0x0F.
W praktyce dla naciśniętego drugiego wiersza ten przykład wygląda następująco:

```
~PIND & 0x0F = ~(0b00001101) & 0b00001111 =
= 0b11110010 & 0b00001111 = 0b00000010 = 2 wiersz
```

W celu sprawdzenia numeru kolumny należy odwrócić stany portów, tzn. zanegować stan DDRD i PORTD w celu ustalenia wierszy jako wyjścia o stanie 0 i kolumn jako wejścia podciągniętego pod stan 1. Wtedy stosuje się powyższe działanie z maską 0xF0 w celu sprawdzenia górnej połowy portu (pod te piny są podpięte kolumny klawiatury). Dla kolumny trzeciej ten przykład wygląda następująco:

```
~PIND & 0xF0 = ~(0b10110000) & 0b11110000 =
= 0b01001111 & 0b11110000 = 0b01000000 = 3 kolumna
```

W celu zamiany numeru kolumny i wiersza z wartości bitowej na dziesiętną korzystam z własnej funkcji **debinary()**, która zwraca numer aktywnego bitu w dolnej połowie bajtu.
Stan klawiatury odczytuję w przerwaniu timera 0 co 1/20 sekundy. Timer działa w trybie CTC i dla wartości preskalera równej 1/1024.
Jeżeli w przerwaniu wykryję naciśnięcie klawisza, wywołuję funkcję **show_number()**, wysyłając do niej numer naciśniętego klawisza liczony z wzoru (<numer wiersza> - 1)*4 + <numer kolumny>), co dla 2 wiersza i 3 kolumny wygląda następująco:

```
(2 -1) *4 + 3 = 1*4 + 3 = 4+3 = 7 klawisz
```

Funkcja **show_number()** sprawdza czy wciśnięty numer klawisza jest już wyświetlony na ekranie poprzez porównanie jego wartości z zmienną globalną **number**. Jeżeli numer klawisza jest inny niż ten na ekranie, funkcja czyści wyświetlacz za pomocą **LCD_CLEAR()**, zamienia numer na ciąg znaków funkcją **itoa()**, a następnie wypisuje go na ekran funkcją **LCD_WRITE()**. Na samym końcu aktualizowana jest wartość zmiennej **number**.
Warto zaznaczyć, że w bibliotece LCD musiałem zmienić jedynie jedną rzecz - podczas inicjalizacji wyświetlacza wyłączam kursor, ponieważ jego miganie rozprasza i utrudnia odczyt z ekranu.

3. Wnioski

Podczas tych laboratoriów nauczyłem się następujących rzeczy:

1. Powtórzyłem wiadomości o przerwaniach w timerze 0, co pozwoliło mi rozwiązać problem związany z zmienną **number**. Jako że ta zmienna jest używana w funkcji wywoływanej podczas obsługi przerwania, muszę uważać na optymalizacje kompilatora, ponieważ mogą one powodować błędy w działaniu, tak jak w poniższym przykładowym kodzie (jest to tylko przykład):

```
int number = 0; // inicjalizacja zmiennej
ISR (TIMER0_COMP_vect) { ... } // obsługa przerwania, zmieniam w nim wartość "number"
int main(){
    ... // inicjalizacja progranu, wywołanie sei()
    number = 5; // ustawiamy nową wartość number
    //w tym momencie jest wywoływane przerwanie i wartość number zmienia się na 10
    number++;  // zwiększamy wartość o 1, czyli powinno być 11 a mamy 6
}
```

   W tym przykładzie kompilator zoptymalizował program przechowując wartość zmiennej **number** w rejestrze procesora pomiędzy operacją przypisania i inkrementacji, zamiast wykonać odczyt i zapis do pamięci. Przez to w momencie wykonania przerwania i zapisania w nim nowej wartości do zmiennej **number** ta wartość jest tracona, ponieważ jest nadpisywania wartością zmiennej **number** zapamiętaną w rejestrze. W celu uniknięcia takich błędów należy użyć słowa kluczowego **volatile** przy inicjalizacji zmiennej w celu wyłączenia jakichkolwiek optymalizacji. Działający przykład wygląda tak:

```
volatile int number = 0; // inicjalizacja zmiennej z wyłączoną optymalizacją
ISR (TIMER0_COMP_vect) { ... } // obsługa przerwania, zmieniam w nim wartość "number"
int main(){
    ... // inicjalizacja progranu, wywołanie sei()
    number = 5; // ustawiamy nową wartość number
    //w tym momencie jest wywoływane przerwanie i wartość number zmienia się na 10
    number++;  // zwiększamy wartość o 1, i teraz na pewno mamy wartość 11
}
```

2. Nauczyłem się obsługiwać klawiaturę za pomocą algorytmu odczytu z odwracaniem kierunków linii w porcie.
    a. Moim głównym problemem było wyodrębnienie numerów naciśniętego wiersza i kolumny. W tym celu wykorzystałem wiadomości z poprzednich laboratoriów na temat maskowania, negacji i wybiórczej negacji oraz przesunięć bitowych.
    b. Okazało się również, że funkcja **itoa()** nie dodaje na końcu bufora znaku kończącego ciąg znaków (null-terminatora), co powodowało zawieszenie się mojego funkcji **LCD_WRITE()**, która oczekiwała zakończenia każdego stringa null-terminatorem. Rozwiązałem to manualnie dodając null-terminator na końcu bufora przekazanego do **itoa()**