



Katedra Systemów Geoinformatycznych

Aplikacje Systemów Wbudowanych

Laboratorium

część 1
ćwiczenia 1-3



Politechnika Gdańska
Wydział Elektroniki, Telekomunikacji i Informatyki
Katedra Systemów Geoinformatycznych

Gdańsk 2010

Spis treści

1	Tematy ćwiczeń.....	3
1.1	Ćwiczenie AVR-1. Implementacja gry logicznej.....	3
1.1.1	Układ laboratoryjny Keyboard + Matrix	3
1.2	Ćwiczenie AVR-2. Model alarmu samochodowego.....	6
1.2.1	Układ laboratoryjny Keyboard + Matrix	6
1.3	Ćwiczenie AVR-3. System sterowania sygnalizatorami.....	9
1.3.1	Panel symulatora skrzyżowania.....	9
2	Programowanie: Język C.....	12
2.1	Moduły biblioteki avr-libc.....	12
2.2	Moduły dedykowane do laboratorium.....	13
2.2.1	Moduł <lcd.h>	13
2.2.2	Moduł <one_wire.h>	14
2.2.3	Przykłady funkcji przydatnych przy realizacji programu	15
2.3	Obsługa przerwań	18

1 Tematy ćwiczeń

1.1 Ćwiczenie AVR-1. Implementacja gry logicznej.

Wykorzystywany sprzęt:

- System uruchomieniowy EVB503 z procesorem ATMega128.
- Układ laboratoryjny Keyboard + Matrix.
- 2 10-przewodowe, taśmowe kable połączeniowe.
- Urządzenie JTAG-TWICE, kabel szeregowy RS232.
- Komputer: środowisko AVR Studio, kompilator avr-gcc.

Cel ćwiczenia:

Zaprogramowanie prostej jedno i dwuosobowej gry planszowej logicznej typu kółko i krzyżyk.

Zadania:

- Napisać program implementujący grę typu kółko i krzyżyk w układzie laboratoryjnym.
- Gra powinna być jedno (drugi ruch mikroprocesora) i dwuosobowa, do wyboru przez użytkownika.
- Należy wykonać wersję 9 i 16 połową do wyboru.
- W wariacie jednoosobowym należy umożliwić wybór kto zaczyna.
- Wskazane jest umożliwienie określenia poziomu losowych błędów algorytmu gry.
- Po wygranej może nastąpić krótka animacja przedstawiająca radość mikroprocesora ze zwycięstwa (lub smutek przegranej)
- Skompilować i uruchomić program.

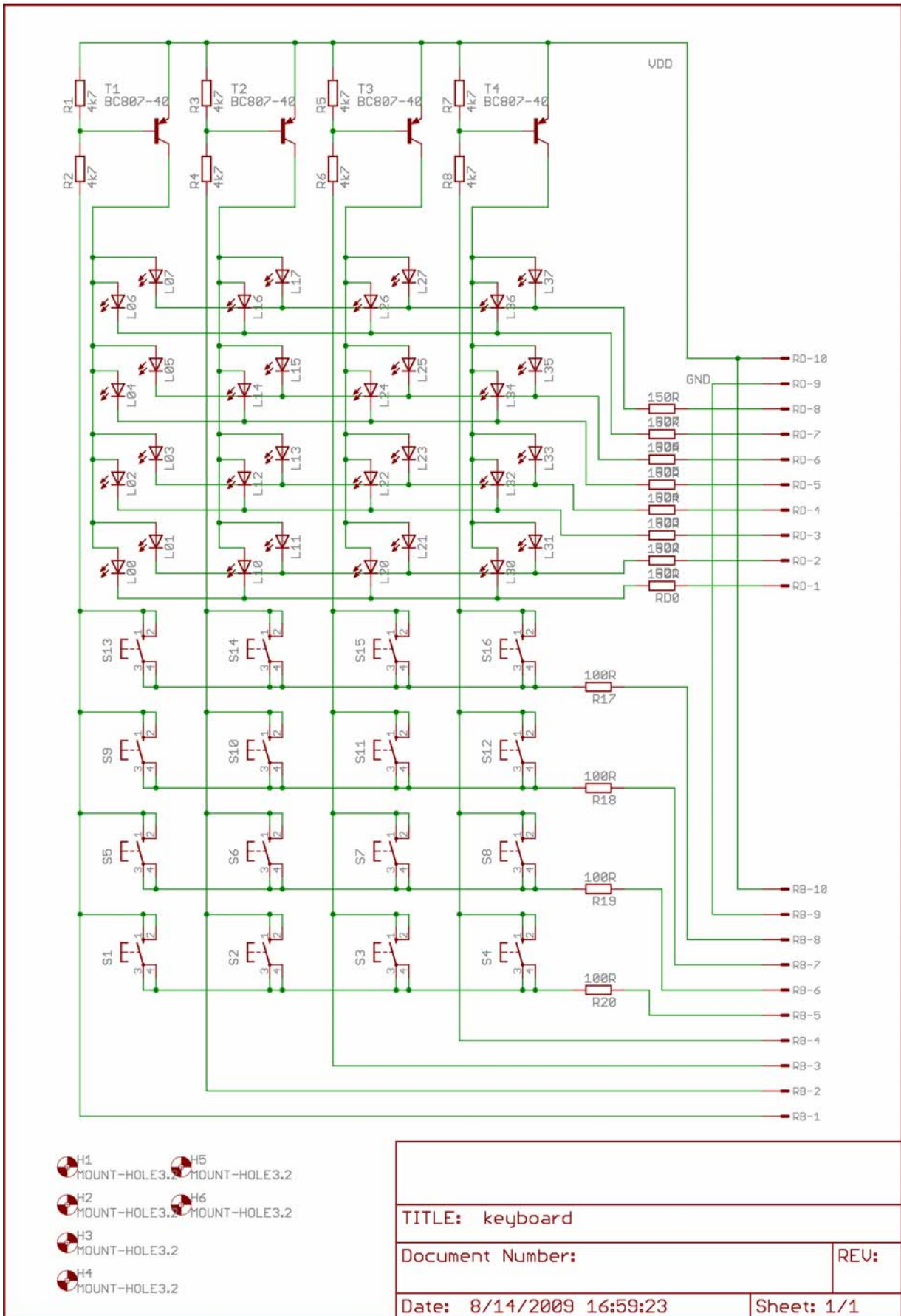
1.1.1 Układ laboratoryjny Keyboard + Matrix

Układ składa się z 16-połowej klawiatury multipleksowanej z 32 diodową matrycą. Diody ustawione są w 16 parach czerwono-zielonych. Posiada 2 złącza RB i RD. Młodsza połowa portu RB – RB0..3 (ustawiona jako wyjście mikrokontrolera) używana jest do wyboru kolumny matrycy klawiatury oraz diod. Wyboru dokonuje się poprzez wystawienie 0 na odpowiedniej linii.

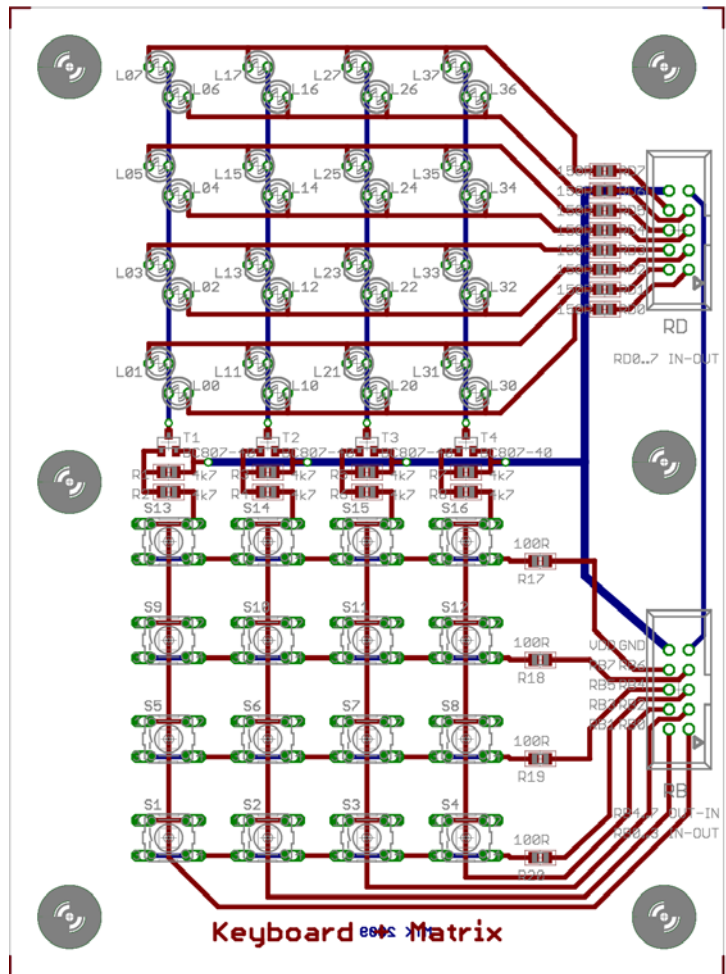
Starsza połowa portu – RB4..7 (ustawiona jako wejście mikrokontrolera) służy do detekcji (pojawi się 0) wciśniętego klawisza z zaadresowanej kolumny.

Cały port RD (wyjście mikrokontrolera) służy do wyboru diody w kolumnie. Stanem aktywnym jest 0.

Na rysunku 1.1 przedstawiono schemat ideowy układu laboratoryjnego Keyboard + Matrix, natomiast na rysunku 1.2 przedstawiono rysunek układu.



Rysunek 1.1. Schemat ideowy układu laboratoryjnego Keyboard + Matrix



Rysunek 1.2. Rysunek układu laboratoryjnego Keyboard + Matrix

1.2 Ćwiczenie AVR-2. Model alarmu samochodowego.

Wykorzystywany sprzęt:

- System uruchomieniowy EVB503 z procesorem ATMega128.
- Układ laboratoryjny Autoalarm Tester.
- 2 10-przewodowe, taśmowe kable połączeniowe.
- Urządzenie JTAG-TWICE, kabel szeregowy RS232.
- Komputer: środowisko AVR Studio, kompilator avr-gcc.

Cel ćwiczenia:

Opracowanie założeń projektowych oraz zaprogramowanie wygodnego do użytkowania alarmu samochodowego.

Zadania:

- Przed rozpoczęciem ćwiczenia należy zapoznać się z istniejącymi rozwiązaniami układów alarmowych do samochodów osobowych.
- W pierwszej części ćwiczenia zapoznać się z możliwościami układu laboratoryjnego, zaplanować (na papierze) funkcje alarmu, przypisać poszczególne funkcje do klawiszy pilota.
- W drugiej części, napisać program alarmu.
- Skompilować i uruchomić program.

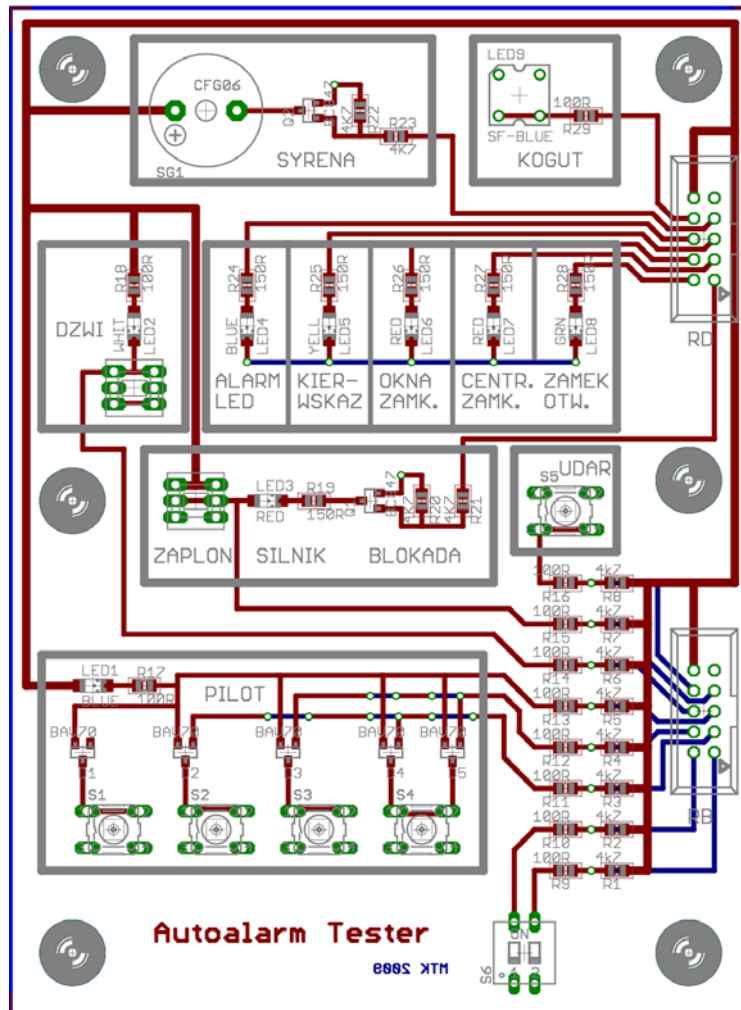
1.2.1 Układ laboratoryjny Keyboard + Matrix

Na dolnej warstwie płytki drukowanej znajduje się płaszczyzna masy.

Na linii PORTB.4 pojawia się stan niski w momencie wciśnięcia dowolnego przycisku pilota. Na liniach PORTB.2..3 pojawia się zakodowany numer przycisku. Na linii PORTB.5 pojawia się stan niski w momencie otwarcia drzwi samochodu. Na linii PORTB.6 stan wysoki przy włączonym silniku, a na PORTB.7 stan niski z czujnika udaru.

Przełącznik S6 służy do ustawienia parametrów alarmu na stałe.

Na rysunku 1.3 przedstawiono schemat ideowy układu laboratoryjnego Autoalarm Tester, natomiast na rysunku 1.4 przedstawiono rysunek układu.



Rysunek 1.4. Rysunek układu laboratoryjnego Autoalarm Tester

1.3 Ćwiczenie AVR-3. System sterowania sygnalizatorami.

Wykorzystywany sprzęt:

- System uruchomieniowy EVB503 z procesorem ATMega128.
- Panel symulatora skrzyżowania ulic.
- Urządzenie JTAG-TWICE, kabel szeregowy RS232.
- Komputer: środowisko AVR Studio, kompilator avr-gcc.

Cel ćwiczenia:

Opracowanie systemu sterowania symulatorem świateł na skrzyżowaniu ulic. Panel symulatora dołączany jest za pośrednictwem portu B mikroprocesora. Przyciski układu uruchomieniowego dołączone są do portu D. Symulator umożliwia sterowanie ruchem na skrzyżowaniu z uwzględnieniem:

- sygnalizatorów głównych (czerwone, pomarańczowe, zielone)
- sygnalizatorów dla pieszych (czerwone, zielone)
- sygnalizatorów dla skręcających w lewo (czerwone, pomarańczowe, zielone)
- sygnalizatorów dla skręcających w prawo (zielone)
- przycisków żądania przejścia dla pieszych.

W programie należy opracować funkcję przesyłającą do symulatora 6-bajtowe kombinacje danych sterujących diodami LED w określonych odstępach czasu.

Zadania:

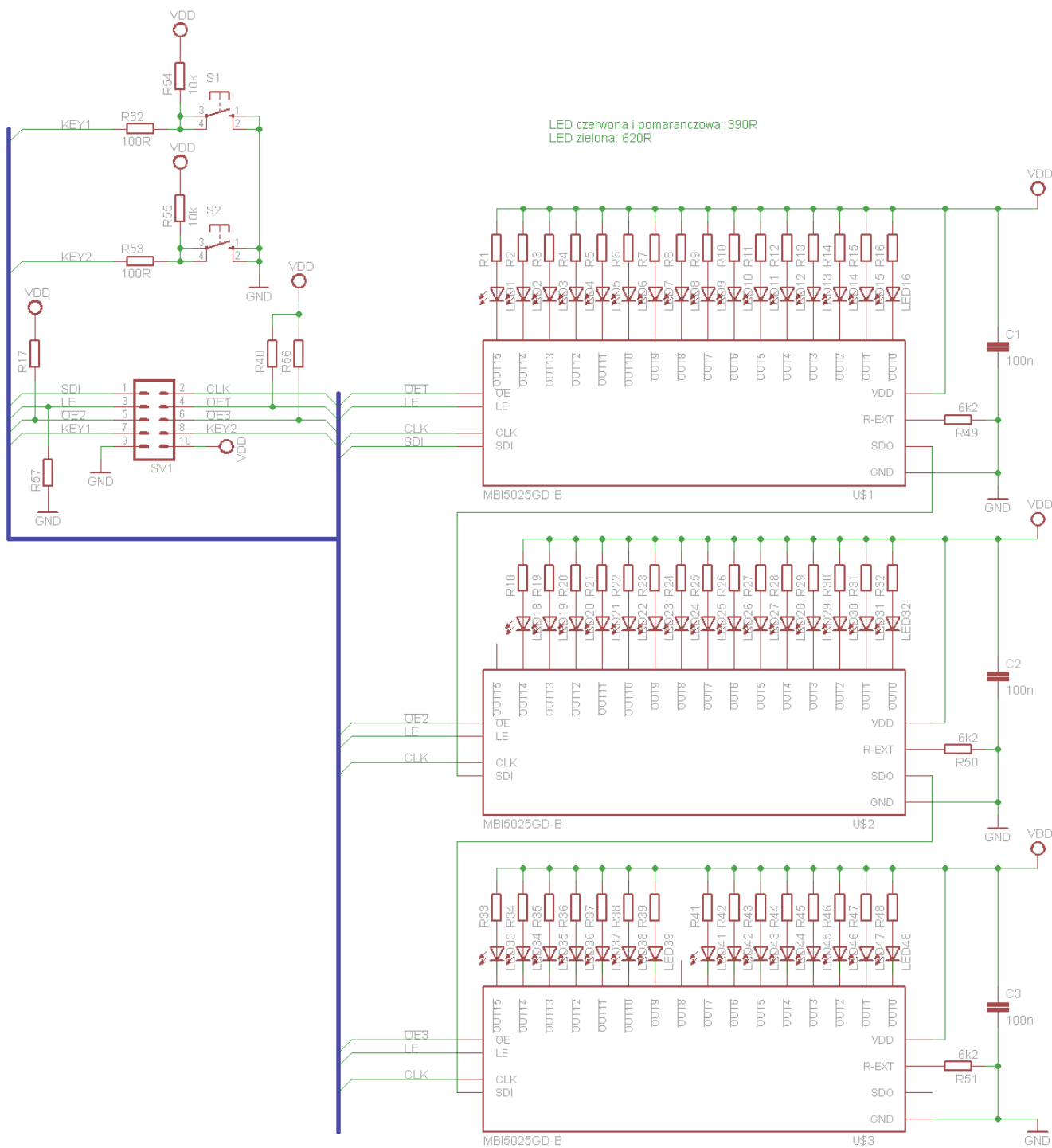
- Złożyć nowy projekt. Program powinien zapewnić możliwość włączenia sygnalizacji awarii (wszystkie światła pomarańczowe cyklicznie zapalają się na 0,5 sekundy i gasną na ten sam okres). Skompilować i uruchomić program. Rozbudować program o sterowanie ruchem za pomocą sygnalizatorów głównych i sygnalizatorów dla pieszych. Zmiana świateł co 10 sekund. Światło pomarańczowe świeci przez 1 sekundę. Przypisać jednemu z przycisków na układzie uruchomieniowym tryb awarii, a drugiemu tryb roboczy.
- Wprowadzić do programu złożone tryby pracy, z wykorzystaniem wszystkich dostępnych elementów panelu symulatora (skręt w lewo, skręt w prawo, przejście dwuetapowe, żądanie przejścia). Przełączanie trybów pracy za pomocą przycisków w układzie uruchomieniowym. Skompilować i uruchomić program.

1.3.1 Panel symulatora skrzyżowania

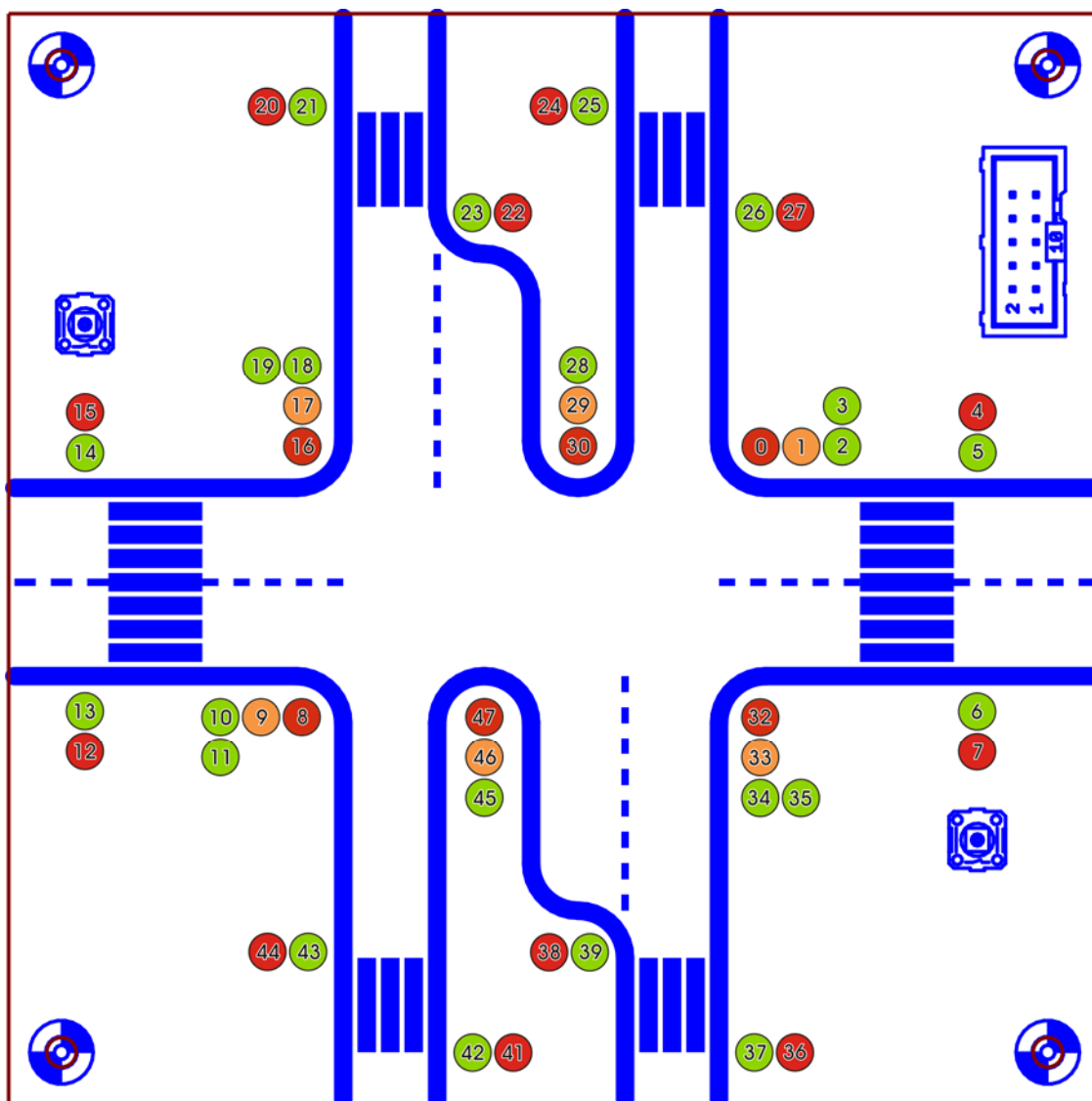
Schemat panelu symulatora przedstawiono na rysunku 1.5. Komunikacja z panelem odbywa się za pomocą jednego z portów procesora (w Laboratorium będzie to PORTB). Dwie najbardziej znaczące linie portu (6 i 7) pracują jako wejścia i zwracają informację na temat stanu przycisków S1 i S2. Pozostałe linie pracują jako wyjścia. Ich przeznaczenie jest następujące:

Nr linii	Opis
0	DATA. Linia szeregowego przesyłania danych do rejestru przesuwanego.
1	CLK. Dane z linii 0 przepisywane są przy narastającym zboczu sygnału zegarowego.
2	LE. Przepisanie danych z rejestru do bufora przy narastającym zboczu sygnału w linii.
3	/OE1. Włączenie diod połączonych z układem 1
4	/OE2. Włączenie diod połączonych z układem 2
5	/OE3. Włączenie diod połączonych z układem 3

Wyświetlenie określonej kombinacji diod wymaga wysłania szeregowo 48 bitów danych (poczynając od najstarszego) do rejestru przesuwanego. Dla przesłania każdego bitu wymagana jest zmiana sygnału zegara w linii 1 z 0 na 1. Wartość "1" bitu danych oznacza świecenie diody. Następnie należy przez zmianę sygnału z 0 na 1 w linii 2 spowodować przepisanie stanu rejestru do buforów. Aby podać napięcie z buforów na diody należy ustawić linie zezwolenia (OE1..OE3) w stan "0".



Rysunek 1.5. Schemat układu symulatora



Rysunek 1.6. Widok panelu symulatora skrzyżowania ulic.

Na rysunku 1.6. przedstawiono rozmieszczenie diod na panelu symulatora wraz z numerami bitów rejestru odpowiedzialnych za ich obsługę.

2 Programowanie: Język C

W trakcie wykonywania ćwiczeń laboratoryjnych należy napisać i uruchomić programy w języku C (używając dostarczonego przez pakiet *WinAVR* zestawu: kompilatora *avr-gcc*, narzędzi *avr-binutils* oraz biblioteki *avr-libc*), realizujące zadane tematy ćwiczeń. Programy narzędziowe dostarczane przez środowisko przetwarzają kod źródłowy C, zapisany w pliku wejściowym na kod wykonawczy, zapisywany w pamięci FLASH mikroprocesora (oraz ewentualnie dane zapisywane w pamięci EEPROM).

W projektach realizowanych w Laboratorium można używać modułów opracowanych dla sprzętu wykorzystywanego w ćwiczeniach (wyświetlacz LCD oraz komunikacja 1-Wire). Aby użyć modułów **lcd** i/lub **one_wire**, należy w ustawieniach kompilatora dodać bibliotekę **libswm** (opcja `-lswm`). Gdy używamy środowiska AVR Studio, należy z menu głównego wybrać **Project.Configuration Options**, a następnie w zakładce **Libraries** dołączyć do projektu zarchiwizowany plik biblioteki **libswm.a**. Takie ustawienie wywoła wprowadzenie zmian do pliku Makefile używanego do sterowania procesem kompilacji.

2.1 Moduły biblioteki avr-libc

Kompilator *avr-gcc* dostarczany jest wraz z dedykowaną biblioteką *avr-libc*, której moduły można dołączać do programu dyrektywą `#include`. Ścieżka dostępu do folderu *include*, zawierającego moduły biblioteki *avr-libc* jest umieszczona w zmiennej środowiskowej PATH podczas instalacji kompilatora. Poniżej przedstawiono wykaz modułów biblioteki:

Folder *include*

<alloca.h>:	Alokacja pamięci dla stosu
<assert.h>:	Diagnostyki
<ctype.h>:	Działania na zmiennych znakowych
<errno.h>:	Błędy systemowe
<inttypes.h>:	Konwersje typów całkowitych
<math.h>:	Funkcje matematyczne
<setjmp.h>:	Skoki nielokalne
<stdint.h>:	Standardowe typy całkowite
<stdio.h>:	Standardowe procedury obsługi wejścia/wyjścia
<stdlib.h>:	Podstawowe procedury obsługi
<string.h>:	Działania na łańcuchach

Podfolder *include/avr*

<avr/boot.h>:	Funkcje obsługi boot-loadera
<avr/eeprom.h>:	Obsługa pamięci EEPROM
<avr/fuse.h>:	Obsługa przelączników (Fuse)
<avr/interrupt.h>:	Przerwania
<avr/io.h>:	Definicje obsługi wejścia/wyjścia, specyficzne dla procesora AVR
<avr/lock.h>:	Obsługa bitów konfiguracyjnych (Lockbit)
<avr/pgmspace.h>:	Obsługa pamięci programu (FLASH)
<avr/power.h>:	Obsługa oszczędzania energii
<avr/sfr_defs.h>:	Definicje rejestrów procesora
<avr/sleep.h>:	Obsługa oszczędzania energii i trybów hibernacji (Sleep)
<avr/version.h>:	Numer wersji biblioteki
<avr/wdt.h>:	Obsługa watchdoga

Podfolder *include/util*

<util/atomic.h>:	Atomically and Non-Atomically Executed Code Blocks
<util/crc16.h>:	Obliczanie CRC
<util/delay.h>:	Funkcje czasowego wstrzymywania programu
<util/delay_basic.h>:	Podstawowe pętle opóźniające
<util/parity.h>:	Generowanie bitów kontroli parzystości
<util/setbaud.h>:	Makra do obliczania prędkości transmisji
<util/twi.h>:	Definicje masek dla transmisji I ² C (TWI)

Podfolder *include/compat*

<compat/deprecated.h>:	Elementy przestarzałe, wycofane z biblioteki
<compat/ina90.h>:	Kompatybilność z IAR EWB 3.x

2.2 Moduły dedykowane do laboratorium

2.2.1 Moduł <lcd.h>

Moduł dołączamy dyrektywą `#include <lcd.h>`. Zawartość pliku nagłówkowego *lcd.h*:

```
//Procedury obsługi wyświetlacza LCD:
#ifndef _LCD_H_
# define _LCD_H_ "lcd.h"
#else
# error "Attempt to include more than one <lcd.h> file."
#endif
#ifndef _AVR_IO_H_
# include <avr/io.h>
#endif
/* Dla obsługi wyświetlacza należy uaktywnić dostęp do zewnętrznej
pamięci RAM. W tym celu ustawiamy bity SRE i SRW10 w rej. MCUCR
oraz bity SRW00, SRW01 i SRW11 w rejestrze XMCRA. Przykładowo może
to być zrealizowane w postaci kodu:
    MCUCR = _BV(SRE) | _BV(SRW10);
    XMCRA = _BV(SRW00) | _BV(SRW01) | _BV(SRW11);
Rejestry wyświetlacza dostępne są pod adresami:
- rejestr sterujący IR      - 0x1F90 (COMM_LCD)
- rejestr danych DR        - 0x1F91 (DATA_LCD) */
#define EXT_MEM8(mem_addr) (*(volatile uint8_t *) (mem_addr))
#define COMM_LCD      EXT_MEM8(0x1F90)
#define DATA_LCD     EXT_MEM8(0x1F91)
#define BF            7      // Bit zajętości wyświetlacza

/* Sprawdzanie zajętości wyświetlacza
Procedura czeka na wyzerowanie flagi BF */
void
test_bf(void);

/* Wysłanie bajtu do IR
Wysłanie rozkazu lub adresu */
void
pisz_com(uint8_t _cmd);

/* Wysłanie bajtu do DR
W zależności od adresu bajt będzie
wysłany do DD RAM lub CG RAM */
void
pisz_ws(uint8_t _data);

/* Czytanie bajtu z DR
Wczytanie bajtu danych spod aktualnego adresu
w DD RAM lub CG RAM */
uint8_t
czyt_ws(void);

/* Czytanie bajtu z IR
Wczytanie aktualnego adresu DD RAM lub CG RAM
i flagi BF na najstarszym bicie */
uint8_t
czyt_ad(void);
```

2.2.2 Moduł <one_wire.h>

Moduł dołączamy dyrektywą `#include <one_wire.h>`. Zawartość plik nagłówkowego `one_wire.h`:

```
// Procedury obsługi układu iButton

#ifndef _ONE_WIRE_H_
# define _ONE_WIRE_H_ "one_wire.h"
#else
# error "Attempt to include more than one <one_wire.h> file."
#endif

#ifndef _AVR_IO_H_
# include <avr/io.h>
#endif

/* Szyna sygnałowa: iB_LINE w porcie _IB_
   Domyślnie: linia 0 w porcie E*/
#ifndef _IB_
# define _IB_      E
# define iB_tx     DDRE
# define iB_wy     PORTE
# define iB_rx     PINE
#else
# define iB_tx     DDR(_IB_)
# define iB_wy     PORT(_IB_)
# define iB_rx     PIN(_IB_)
#endif

#ifndef iB_LINE
# define iB_LINE   0
#endif

/* Nadanie bitu "0" lub "1" w linii
   We: bit.0 = nadawany bit */
void
iB_send_bit(char _bit);

/* Odbiór bitu z linii
   Wy: return.0 = odebrany bit */
char
iB_recv(void);

/* Rozkaz Reset 1-Wire
   Wy: 0x00 - jest pastylka
       0x01 - brak pastylki
       0xFF - zwarcie w linii */
char
iB_reset(void);
```

2.2.3 Przykłady funkcji przydatnych przy realizacji programu

```
/* Dodatkowe procedury obsługi wyświetlacza LCD, wymagają dołączenia
modułu <lcd.h>.
Rejestry wyświetlacza dostępne są pod adresami:
- rejestr sterujący 0x1F90 (COMM_LCD)
   char *COMM_LCD = 0x1F90;
- rejestr danych    0x1F91 (DATA_LCD)
   char *DATA_LCD = 0x1F91; */

#ifdef __PGMSPACE_H_
# include <avr/pgmspace.h>
#endif

// Zerowanie wyświetlacza
void
lcd_clear(void);

// Przesunięcie kursora
void
lcd_home(void);

// Inicjalizacja wyświetlacza 5x7, 2 wiersze
void
init_lcd(void);

// Przepisanie tekstu (koniec = 0xFF) z pamięci programu na wyświetlacz
// Funkcja dopuszcza wykorzystanie znaku o kodzie 0x00
// _adres - adres tekstu w pam. FLASH
void
disp_txt_P(const char* _adres);

// Przepisanie tekstu (koniec = 0x00) z pamięci danych na wyświetlacz
// _adres - adres tekstu w pam. RAM
void
disp_txt_D(char* _adres);
//Implementacja

void
lcd_clear(void)
{
    pisz_com(0x01);
}

void
lcd_home(void)
{
    pisz_com(0x02);
}

void
init_lcd(void)
{
    _delay_ms(15);
    COMM_LCD = 0x30;
    _delay_ms(5);
    COMM_LCD = 0x30;
    _delay_ms(0.2);
    COMM_LCD = 0x30;
    _delay_ms(30);
    COMM_LCD = 0x38; //Słowo danych 8-bitów, dwa wiersze, znak 7x5 pikseli
    test_bf();
}
```

```

    pisz_com(0x0C); //Włączenie wyświetlacza, bez kursora, bez migotania
    lcd_clear();
    pisz_com(0x06); //Wpisywanie znaków od lewej, autoinkrementacja
    lcd_home();
}

void
disp_txt_P(const char* _adres)
{
    volatile uint8_t al;
    for (int i = 0; i<16; i++)
    {
        al = pgm_read_byte(&_adres[i]);
        if (al == 0xFF) break;
        pisz_ws(al);
    }
}

void
disp_txt_D(char* _adres)
{
    volatile uint8_t al;
    for (int i = 0; i<16; i++)
    {
        al = _adres[i];
        if (al == 0x00) break;
        pisz_ws(al);
    }
}

/* Procedury obsługi iButton w protokole 1-Wire
Wymagają dołączenia modułu <one_wire.h> */

// Wysłanie rozkazu w linii 1-Wire
void
iB_send_cmd(char cmd);

// Inicjacja 1-Wire
// Diody zgaszone
// Linia w stanie wysokim (WE)
void
iB_init(void);

// Rozkaz 33H (Read Rom)
// Odebrane dane w tablicy Tab
// Wy: CRC8 (0 - OK)
char
iB_33h(char* Tab);

// Obsługa LED-ów
#define iB_RED      1
#define iB_GREEN    2

void
iB_zapal(char led);

void
iB_zgas(char led);

// Implementacja

#ifndef _UTIL_CRC16_H
# include <util\crc16.h>

```

```

#endif
void
iB_send_cmd(char cmd)
{
    for (int i = 0; i < 8; i++)
    {
        iB_send_bit(cmd & 0x01);
        cmd >>= 1;
    }
}

void
iB_init(void)
{
    // Diody: kier. WY
    iB_tx |= _BV(iB_RED) | _BV(iB_GREEN);
    // Linia 1-Wire: kier. WE (stan HIGH)
    iB_tx &= ~_BV(iB_LINE);
    // Gasi diody
    iB_zgas(iB_RED);
    iB_zgas(iB_GREEN);
    // Linia 1-Wire: przygotowanie stanu LOW dla nadawania
    iB_wy &= ~_BV(iB_LINE);
}

// READ ROM
char
iB_33h(char* Tab)
{
    volatile char al;
    iB_send_cmd(0x33);
    char crc = 0;
    for (int i = 7; i > -1; i--) // 8 bajtów, od LSB
    {
        al = 0;
        for (int k = 0; k < 8; k++) // 8 bitów, od najmłodszego
        {
            al >>= 1;
            al |= iB_recv() << 7; // Bit na pozycji 7
        }
        crc = _crc_ibutton_update(crc, al);
        Tab[i] = al;
    }
    return(crc);
}

void
iB_zgas(char led)
{
    iB_wy &= ~_BV(led);
}

void
iB_zapal(char led)
{
    iB_wy |= _BV(led);
}

```

2.3 Obsługa przerw

Procedurę obsługi przerwania budujemy za pomocą makra `ISR(vect)`, gdzie *vect* jest nazwą przerwania, zgodnie z poniższą tabelą:

Nr	Description	Vector name	Old vector name
1	External Interrupt Request 0	INT0_vect	SIG_INTERRUPT0
2	External Interrupt Request 1	INT1_vect	SIG_INTERRUPT1
3	External Interrupt Request 2	INT2_vect	SIG_INTERRUPT2
4	External Interrupt Request 3	INT3_vect	SIG_INTERRUPT3
5	External Interrupt Request 4	INT4_vect	SIG_INTERRUPT4
6	External Interrupt Request 5	INT5_vect	SIG_INTERRUPT5
7	External Interrupt Request 6	INT6_vect	SIG_INTERRUPT6
8	External Interrupt Request 7	INT7_vect	SIG_INTERRUPT7
9	Timer/Counter2 Compare Match	TIMER2_COMP_vect	SIG_OUTPUT_COMPARE2
10	Timer/Counter2 Overflow	TIMER2_OVF_vect	SIG_OVERFLOW2
11	Timer/Counter1 Capture Event	TIMER1_CAPT_vect	SIG_INPUT_CAPTURE1
12	Timer/Counter1 Compare Match A	TIMER1_COMPA_vect	SIG_OUTPUT_COMPARE1A
13	Timer/Counter Compare Match B	TIMER1_COMPB_vect	SIG_OUTPUT_COMPARE1B
14	Timer/Counter1 Overflow	TIMER1_OVF_vect	SIG_OVERFLOW1
15	Timer/Counter0 Compare Match	TIMER0_COMP_vect	SIG_OUTPUT_COMPARE0
16	Timer/Counter0 Overflow	TIMER0_OVF_vect	SIG_OVERFLOW0
17	SPI Serial Transfer Complete	SPI_STC_vect	SIG_SPI
18	USART0, Rx Complete	USART0_RX_vect	SIG_U(S)ART0_RECV
19	USART0 Data Register Empty	USART0_UDRE_vect	SIG_U(S)ART0_DATA
20	USART0, Tx Complete	USART0_TX_vect	SIG_U(S)ART0_TRANS
21	ADC Conversion Complete	ADC_vect	SIG_ADC
22	EEPROM Ready	EE_READY_vect	SIG_EEPROM_READY
23	Analog Comparator	ANALOG_COMP_vect	SIG_COMPARATOR
24	Timer/Counter1 Compare Match C	TIMER1_COMPC_vect	SIG_OUTPUT_COMPARE1C
25	Timer/Counter3 Capture Event	TIMER3_CAPT_vect	SIG_INPUT_CAPTURE3
26	Timer/Counter3 Compare Match A	TIMER3_COMPA_vect	SIG_OUTPUT_COMPARE3A
27	Timer/Counter3 Compare Match B	TIMER3_COMPB_vect	SIG_OUTPUT_COMPARE3B
28	Timer/Counter3 Compare Match C	TIMER3_COMPC_vect	SIG_OUTPUT_COMPARE3C
29	Timer/Counter3 Overflow	TIMER3_OVF_vect	SIG_OVERFLOW3
30	USART1, Rx Complete	USART1_RX_vect	SIG_U(S)ART1_RECV
31	USART1, Data Register Empty	USART1_UDRE_vect	SIG_U(S)ART1_DATA
32	USART1, Tx Complete	USART1_TX_vect	SIG_U(S)ART1_TRANS
33	2-wire Serial Interface	TWI_vect	SIG_2WIRE_SERIAL
34	Store Program Memory Read	SPM_READY_vect	SIG_SPM_READY

Tabela 2.1 Wykaz nazw wektorów przerw procesora ATmega-128

Uruchomienie obsługi przerw odbywa się za pomocą makra `sei()` i ustawienia odpowiednich bitów masek we właściwych rejestrach, natomiast globalne zablokowanie systemu obsługi przerw zapewnia makro `cli()`. Dla obsługi przerw należy włączyć do programu moduł biblioteczny `<avr/interrupt.h>`.