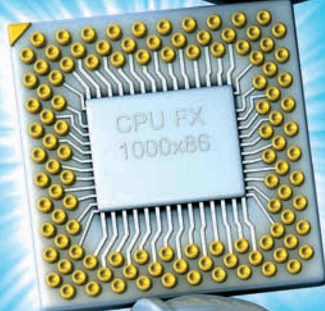
Paweł Borkowski

AVR & ARM7

PROGRAMOWANIE MIKROKONTROLERÓW DLA KAŻDEGO



Poznaj sposoby programowania mikrokontrolerów — nigdy nie wiadomo, kiedy życie zmusi Cię do skonstruowania robota

Jak efektywnie nauczyć się programowania mikrokontrolerów?
Jak skonstruować programator lub zdobyć go w inny sposób?
Jak obsługiwać wyświetlacz LED w czterech językach?

helion.pl

Idź do

• Spis treści
• Przykładowy rozdział

Katalog książek

• Katalog online
• Zamów drukowany katalog

Twój koszyk

• Dodaj do koszyka

Cennik i informacje

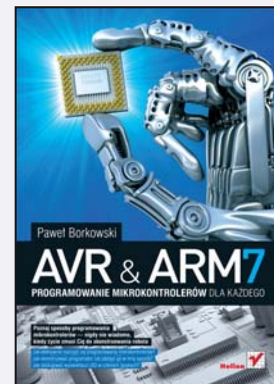
• Zamów informacje o nowościach
• Zamów cennik

Czytelnia

• Fragmenty książek online

AVR i ARM7. Programowanie mikrokontrolerów dla każdego

Autor: Paweł Borkowski
ISBN: 978-83-246-2628-1
Format: 158×235, stron: 528



Poznaj sposoby programowania mikrokontrolerów – nigdy nie wiadomo, kiedy życie zmusi Cię do skonstruowania robota

- Jak efektywnie nauczyć się programowania mikrokontrolerów?
- Jak skonstruować programator lub zdobyć go w inny sposób?
- Jak obsługiwać wyświetlacz LED w czterech językach?

Jeżli nie masz pojęcia o programowaniu mikrokontrolerów, a chcesz się tego nauczyć, ta książka jest właśnie dla Ciebie. Nie musisz wcześniej mieć wiedzy z zakresu elektroniki, ponieważ wszystkie potrzebne pojęcia zostały tu wyjaśnione od podstaw. Niepotrzebna Ci także znajomość programowania w jakimkolwiek języku – te informacje, podane w możliwie najbardziej przystępny sposób, też znajdziesz w podręczniku. Wobec tego wszystko, czego potrzebujesz, to chęć nauki. I jeszcze jedno: może zastanawiasz się, co począć z takim mikrokontrolerem? Otóż możesz zastosować go do konstruowania efektów świetlnych z diod, sterowania modelami samolotów, a nawet sterowania robotami.

Jeżeli wiesz już co nieco na temat programowania mikrokontrolerów, ale chcesz poszerzyć swoją wiedzę – do tego również przyda się ta książka. Dzięki niej dowiesz się, na czym polega programowanie mikrokontrolerów dwóch rodzin: AVR (na przykładzie układu ATmega8) i ARM7 (na przykładzie układu LPC2106). Nauczysz się programowania układów w czterech językach programowania: asemblerze (środowisko AVR Studio 4), języku C (środowisko WinAVR), języku bascom (środowisko Bascom) oraz Pascalu (środowisko mikroPascal). Z łatwością zdobędziesz, a potem – wykonując poszczególne ćwiczenia – sprawdzisz nowe, niesamowite umiejętności, ponieważ cała wiedza podana jest tu przejrzyście i w dodatku z humorem.

- Programowanie mikrokontrolerów z rodziny AVR oraz ARM7
- Obsługa diod i wyświetlaczy LED
- Obsługa przycisków i klawiatur
- Wyświetlacze alfanumeryczne
- Obsługa przerwań
- Komunikacja między mikrokontrolerami (USART)
- Obsługa wyświetlaczy graficznych z telefonu komórkowego Siemens S65
- Serwomechanizmy
- Kompilatory
- Programowanie z użyciem systemów czasu rzeczywistego na przykładzie FreeRTOS

Cała wiedza potrzebna, aby zostać ekspertem od programowania mikrokontrolerów!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2010

Spis treści

Wstęp 7

Poszukiwacze zaginionych portów, czyli jak zacząć przygodę z mikrokontrolerami 9

Część I Programowanie mikrokontrolerów z rodziny AVR 13

Lekcja 1. Instalacja oprogramowania 15
- 1.1. Kompilatory 15
  - 1.1.1. AVR Studio 15
  - 1.1.2. WinAVR 17
  - 1.1.3. Bascom 18
  - 1.1.4. MikroPascal for AVR 20
- 1.2. Programy ładujące 21
  - 1.2.1. PonyProg2000 21
  - 1.2.2. AVRdude 23

Lekcja 2. Cztery i pół metody zdobycia programatora 27
- 2.1. Sample Electronics cable programmer — programator podłączany do portu LPT 27
- 2.2. SI Prog — programator podłączany do portu COM 28
  - 2.2.1. Montaż programatora 28
  - 2.2.2. Montaż adaptera 34
  - 2.2.3. Konfiguracja PonyProg2000 37
- 2.3. USBasp — programator podłączany do portu USB 37
  - 2.3.1. Montaż programatora 37
  - 2.3.2. Podłączanie USBasp do komputera (system Windows) 44
  - 2.3.3. Praca USBasp z AVRdude 46
  - 2.3.4. Praca USBasp z AVR Studio 46
  - 2.3.5. Praca USBasp ze środowiskiem Bascom 47
  - 2.3.6. Praca USBasp z pakietem WinAVR 48
- 2.4. USBasp — zakup kontrolowany 49
- 2.5. Pół metody zdobycia programatora 50
- 2.6. Jak zaprogramować pozostałe układy AVR? 50

Lekcja 3. Zaświecenie diody LED 53
- 3.1. Asembler 55
- 3.2. Język C 62
- 3.3. Bascom 65
- 3.4. Pascal 68
- 3.5. Ćwiczenia 71

Lekcja 4. Mruganie diody LED 73
4.1. Asembler 73
4.2. Język C 79
4.3. Bascom 83
4.4. Pascal 85
4.5. Ćwiczenia 86

Lekcja 5. Obsługa wyświetlacza LED 89
5.1. Asembler 91
5.2. Język C 106
5.3. Bascom 111
5.4. Pascal 114
5.5. Ćwiczenia 118

Lekcja 6. Obsługa przycisku 119
6.1. Asembler 127
6.2. Język C 132
6.3. Bascom 135
6.4. Pascal 138
6.5. Ćwiczenia 141

Lekcja 7. Obsługa klawiatury 143
7.1. Asembler 146
7.2. Język C 159
7.3. Bascom 165
7.4. Pascal 170
7.5. Ćwiczenia 176

Lekcja 8. Obsługa przerwań, a przy tym o bitach konfiguracyjnych i śpiochach słów parę 179
8.1. Asembler 191
8.2. Język C 204
8.3. Bascom 210
8.4. Pascal 217
8.5. Ćwiczenia 223

Lekcja 9. Obsługa wyświetlacza alfanumerycznego LCD 225
9.1. Asembler 229
9.2. Język C 251
9.3. Bascom 264
9.4. Pascal 269
9.5. Ćwiczenia 275

Lekcja 10. ...a zakończą część pierwszą dwa słowa: USART, EEPROM... 277
10.1. Asembler 279
10.2. Język C 293
10.3. Bascom 298
10.4. Pascal 304
10.5. Ćwiczenia 309

Część II Programowanie mikrokontrolerów z rdzeniem ARM7 311

Lekcja 11. Instalacja oprogramowania, przygotowanie oprzyrządowania 313
11.1. Instalacja środowisk programistycznych Keil uVision3 i WinARM oraz programu ładującego Flash Magic 314
11.2. Opis zestawu uruchomieniowego ARE0068 317

Lekcja 12. Igraszki z diodami LED 321
12.1. Język C 324
12.2. Asembler 337
12.3. Ćwiczenia 358
Lekcja 13. Obsługa przycisków 359
13.1. Język C 361
13.2. Asembler 369
13.3. Ćwiczenia 385
Lekcja 14. Przerwania sprzętowe 387
14.1. Język C 392
14.2. Asembler 398
14.3. Ćwiczenia 408
Lekcja 15. Obsługa wyświetlacza graficznego z telefonu Siemens S65. Część 1. 411
15.1. Język C 415
15.2. Asembler 431
15.3. Ćwiczenia 439
Lekcja 16. Obsługa wyświetlacza graficznego z telefonu Siemens S65. Część 2. 441
16.1. Język C 443
16.2. Asembler 457
16.3. Ćwiczenia 464
Lekcja 17. Serwomechanizmy w lewo zwrot, czyli jak zaprogramować ruch robota 467
17.1. Język C 471
17.2. Asembler 482
17.3. Ćwiczenia 488
Lekcja 18. Mały krok w kierunku systemów czasu rzeczywistego — FreeRTOS 491
Skorowidz 513

Lekcja 15

Obsługa wyświetlacza graficznego z telefonu Siemens S65. Część 1.

Temat lekcji to nie pomyłka: rzeczywiście nauczymy się wyświetlać dane na kolorowym wyświetlaczu LCD, używanym w telefonach firmy Siemens serii S65. Podobno informacja o tym fakcie lotem błyskawicy obiegła cały świat, także ten wodny, plotka zatacza coraz szersze kręgi, także na wodzie. Opowiadał mi znajomy ichtiolog… A zresztą zobaczmy, jakie zdjęcia ów ichtiolog przyniósł mi z jeziora (patrz rysunek 15.1).

Ślimak ma rację, mówiąc, że do podłączenia wyświetlacza S65 do mikrokontrolera najlepiej nadaje się interfejs SPI. Jednak dobrze też jest znać alternatywny sposób obsługi wyświetlacza LCD, dlatego w pierwszej kolejności wysyłanie danych do wyświetlacza zaimplementujemy programowo. Do podłączenia wyświetlacza użyjemy linii od P0.16 do P0.21.

Przypomnijmy sobie, jak wygląda moduł wyświetlacza graficznego S65 (patrz rysunek 15.2).

Właściwie gdy piszemy *wyświetlacz graficzny S65*, posługujemy się skrótem pełnej nazwy *wyświetlacz graficzny używany w telefonach Siemens S65*. Aby nie potknąć się o własne nogi, będziemy używać nazw jeszcze krótszych: wyświetlacz S65, wyświetlacz LCD, kolorowy wyświetlacz lub po prostu wyświetlacz. Ponieważ nie będziemy zajmować się innymi wyświetlaczami graficznymi, groźba pomyłki nie istnieje.

Jedyną dostępną pomocą dotyczącą programowania wyświetlacza S65 jest słynny tutorial Christiana Kranza[1]. Możemy się z niego dowiedzieć, że interesujący nas wyświetlacz był wykorzystywany w telefonach firmy Siemens serii S65, M65, CX65 oraz SK65. Moduł ARE0055 zawiera wyświetlacz S65 ze sterownikiem serii LS020xxx firmy Sharp. Matryca wyświetlacza ma rozdzielczość 176×132 pikseli, przy możliwości wyświetlania 16-bitowych kolorów (65 536 kolorów).

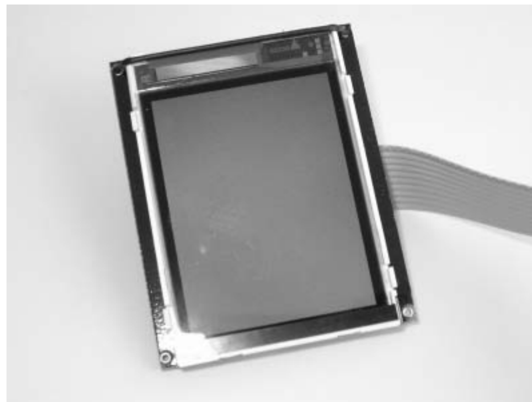
Oryginalnie wyświetlacz ma 10 wyprowadzeń z wejściami na dwa poziomy napięć. Dzięki zastosowanym w module ARE0055 układom elektronicznym liczbę tę udało się zredukować do ośmiu linii zawierających wyprowadzenie zasilania +3.3 V. Ich rozkład na wtyczce modułu przedstawiono na rysunku 15.3.

---

[1] Christian Kranz, *Using the Siemens S65-Display*, *http://www.superkranz.de/christian/S65_Display/DisplayIndex.html*.

Rysunek 15.1. *Opowieść z cyklu Tajemnice podwodnego świata*[2]

Rysunek 15.2. *Moduł wyświetlacza graficznego S65 (ARE0055)*

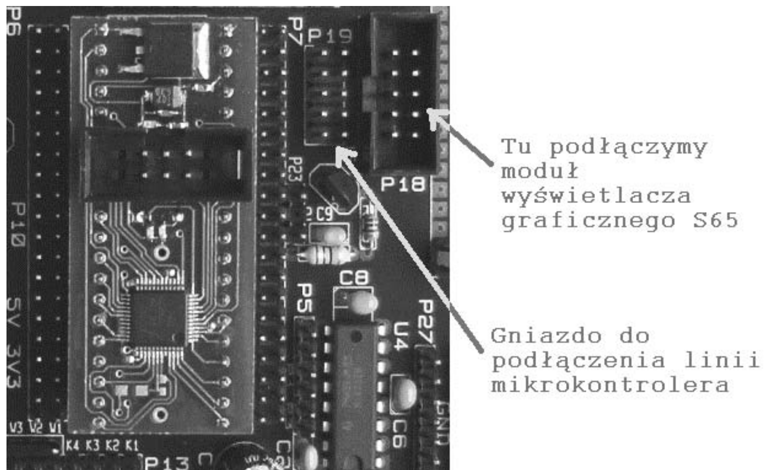


Rysunek 15.3. *Rozkład wyprowadzeń na wtyczce modułu ARE0055 (widok wtyczki z przodu). Oznaczenie NP wskazuje na wyprowadzenie niepodłączone*

RST	• •	RS
CLK	• •	CS
+3V3	• •	DAT
BL	• •	GND
NP	• •	NP

[2] Myślę, że Puccini się nie obrazi, że słowa o Apokalipsie wziąłem — oczywiście — z *Cyganerii*.

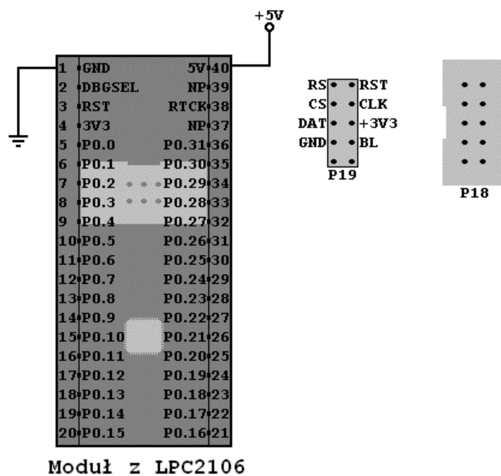
Skoro jest wtyczka, to niebezpodstawne będzie pytanie o miejsce, gdzie można ją wetknąć. Przewidziano do tego celu jedno wspaniałe miejsce na płytce edukacyjnej AE0061. Na rysunku 15.4 widzimy część płytki edukacyjnej, na której uwidocznione zostały gniazda służące do podłączenia modułu ARE0055.



Rysunek 15.4. *Zdjęcie fragmentu płytki edukacyjnej ARE0061 z uwidocznionymi gniazdami służącymi do podłączenia wyświetlacza S65*

Na zdjęciu widzimy dwa gniazda oznaczone P18 i P19. Piny znajdujące się na tych samych miejscach dwóch gniazd są ze sobą połączone. Na przykład pin znajdujący się w lewym górnym rogu gniazda P19 jest połączony z pinem znajdującym się w lewym górnym rogu gniazda P18. Gniazdo P18 służy do podłączenia wtyczki modułu ARE0055, tej wtyczki, której rozkład wyprowadzeń widzieliśmy na rysunku 15.3. Natomiast do pinów gniazda P19 będziemy kablami podłączać wyprowadzenia mikrokontrolera.

Do zilustrowania połączeń układu nie będziemy używać zdjęcia, lecz schematu, który widać na rysunku 15.5.



Rysunek 15.5. *Schemat obszaru płytki edukacyjnej służącego podłączeniu modułu wyświetlacza graficznego S65*

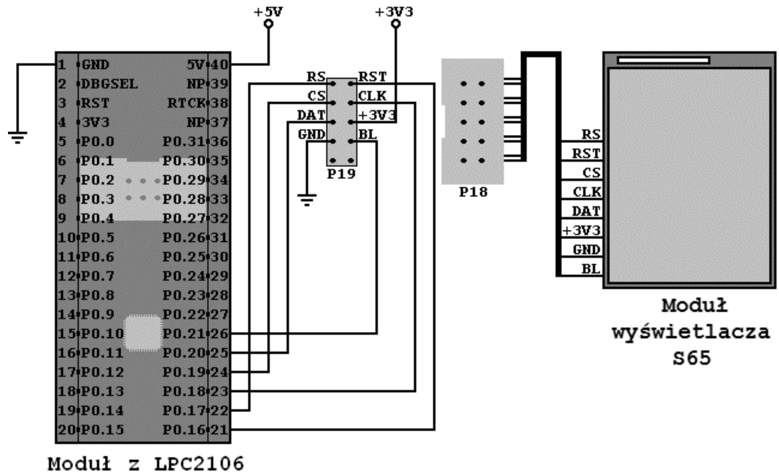
Na schemacie odpowiednimi nazwami oznaczono piny, które po podłączeniu do gniazda P18 modułu ARE0055 będą odpowiadały liniom o tych samych nazwach wyświetlacza LCD.

Zgodnie z zapowiedzią w pierwszym zadaniu interfejs komunikacji między mikrokontrolerem a wyświetlaczem S65 zostanie obsłużony programowo. Wykorzystamy przy tym konfigurację połączenia mikrokontrolera i wyświetlacza opisaną w tabeli 15.1.

Tabela 15.1. *Konfiguracja połączenia mikrokontrolera LPC2106 i modułu wyświetlacza S65*

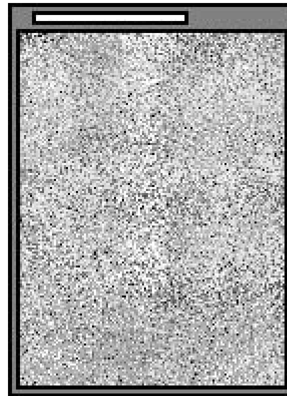
Wyprowadzenie mikrokontrolera	Linia modułu wyświetlacza S65
P0.16	RST
P0.17	RS
P0.18	CLK
P0.19	CS
P0.20	DAT
P0.21	BL

Należy zaznaczyć, że przy programowej implementacji interfejsu komunikacji sposób połączenia mikrokontrolera z modułem wyświetlacza może być dowolny. Schemat przedstawionej konfiguracji połączeń widać na rysunku 15.6. Będzie to nasz oficjalny schemat, który wykorzystamy do rozwiązania pierwszego zadania w ramach tej lekcji.



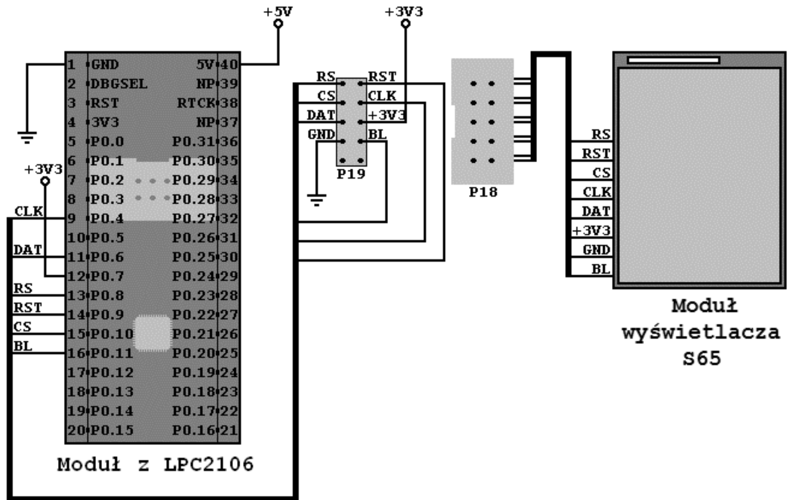
Rysunek 15.6. *Schemat układu do pierwszego zadania — obsługi wyświetlacza graficznego bez interfejsu SPI*

Zarówno w pierwszym, jak i w drugim zadaniu chodzi wyłącznie o poprawne zainicjowanie wyświetlacza S65. Efektem tego powinien być kolorowy szum na ekranie wyświetlacza (patrz rysunek 15.7).



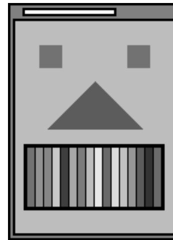
Rysunek 15.7. *Postmodernistyczny szum na wyświetlaczu — sygnał poprawnie zainicjowanego wyświetlacza graficznego*

Do rozwiązania drugiego zadania użyjemy interfejsu SPI, dzięki czemu nasz sposób komunikacji z modułem wyświetlacza graficznego stanie się w pełni profesjonalny. Schemat układu, który wykorzystamy przy rozwiązywaniu drugiego zadania, przedstawiono na rysunku 15.8.



Rysunek 15.8. *Schemat układu do drugiego zadania — obsługi wyświetlacza graficznego poprzez interfejs SPI*

W trzecim zadaniu pokusimy się o zaprogramowanie rysunku, który swoim blaskiem opromieni nas na lata, a krytykom sztuki odbierze ich krytyczny oręż. Narysujemy bowiem bohaterską twarz pana Ziutka (patrz rysunek 15.9).



Rysunek 15.9. *Trzecie zadanie, czyli twarz pana Ziutka*

Zauważmy, że zęby pana Ziutka, zgodnie zresztą ze stanem faktycznym, będą mieniły się w 16 podstawowych kolorach: od błękitu (5 odcieni), poprzez zieleń (6 odcieni), aż do czerwieni (5 odcieni).

Znamy zadania, więc do dzieła!

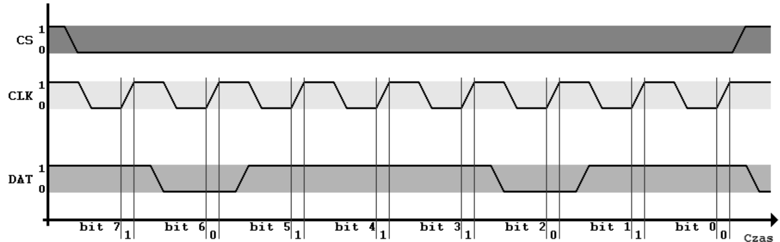
15.1. Język C

Zagadnienia:

- ♦ **Wysyłanie danych do sterownika wyświetlacza S65.**
- ♦ **Inicjalizacja wyświetlacza S65.**
- ♦ **Interfejs SPI.**
- ♦ **Nowy odcinek opowieści dydaktycznej *Sekretny świat misia Ekrysia*, czyli łączmy P0.7 z 3V3.**
- ♦ **16-bitowe kodowanie kolorów.**
- ♦ **Rysowanie prostokątnych obszarów.**

Zaznajamianie się z obsługą wyświetlacza S65 zaczniemy od omówienia roli linii łączących wyświetlacz z mikrokontrolerem. Do wysyłania danych do sterownika wyświetlacza służy linia DAT. Stan wysoki na linii oznacza wysyłanie wartości 1, stan niski to nic innego, jak wysłanie wartości 0. Oczywiście musimy poinformować sterownik wyświetlacza, że ustawienie linii DAT jest znaczące. Robi się to poprzez narastające zbocze CLK (patrz rysunek 15.10).

Rysunek 15.10.
Schemat obsługi linii CS, CLK i DAT przy wysyłaniu do sterownika wyświetlacza S65 bajtu danych



Dzięki doskonałemu rysunkowi 15.10 uściślimy wiadomości dotyczące wysyłania danych do sterownika wyświetlacza S65:

1. Dane wysyłamy w paczkach po 8 bitów.
2. Wysyłanie bajtu danych jest sygnalizowane stanem niskim linii CS..
3. Dane wysyłane są linią DAT w kolejności od najstarszego do najmłodszego bitu.
4. Wysłanie znaczącego bitu danych zostaje potwierdzone narastającym zboczem linii CLK.

Aż tyle wywnioskowaliśmy z jednego małego rysunku? O tak. Rysunek 15.10 należy do naprawdę wspaniałych ilustracji. Bystry obserwator łatwo zauważy w nim także liczby, które dadzą najwyższą wygraną w najbliższym losowaniu Lotto. A tymczasem spróbujmy poznane wiadomości o wysyłaniu danych przekuć we fragment kodu.

W pierwszej kolejności zdefiniujemy połączenie linii mikrokontrolera i wyświetlacza. Zgodnie z rysunkiem 15.6 powinno ono wyglądać tak:

```
#define S65_RST 1<<16
#define S65_RS  1<<17
#define S65_CLK 1<<18
#define S65_CS  1<<19
#define S65_DAT 1<<20
#define S65_BL  1<<21
```

Przypominam, że w pierwszym zadaniu nie używamy interfejsu SPI. Załóżmy, że definiujemy funkcję o następującej deklaracji:

```
void Wyslij_dane_S65(int8 dane);
```

Jej zadaniem będzie wysłanie bajtu danych, danego parametrem `dane`, do sterownika wyświetlacza. W opisie argumentu podaliśmy nową nazwę typu utworzoną instrukcją `typedef`.

```
typedef unsigned char int8;
```

Z rysunku 15.10 wiemy, że wysyłanie danych musi być poprzedzone wyzerowaniem linii CS. Zakończenie wysyłania danych zasygnalizujemy wysokim stanem CS. Tak oto rysuje nam się podstawowy szkielet funkcji `Wyslij_dane_S65`:

```
void Wyslij_dane_S65(int8 dane)
{
  //sygnalizuj wysyłanie danych (poziom niski CS)
    IOCLR = S65_CS;

  ////////////////////////////////////////////////
  //tu będzie kod wysyłania danych
  ////////////////////////////////////////////////
```

```
  //sygnalizuj koniec wysyłania danych (poziom wysoki CS)
    IOSET = S65_CS;
}
```

Wiemy, że wysyłanie danych musi odbywać się od bitu najstarszego do najmłodszego. Najprostszym, a jednocześnie najszybszym sposobem realizacji tego zamiaru będzie kolejne testowanie bitów argumentu `dane`. Na przykład tak wyślemy bit siódmy i szósty:

```
  //bit 7
    if(dane & 0x80) IOSET = S65_DAT; else IOCLR = S65_DAT;
    IOCLR = S65_CLK;
    IOSET = S65_CLK;

  //bit 6
    if(dane & 0x40) IOSET = S65_DAT; else IOCLR = S65_DAT;
    IOCLR = S65_CLK;
    IOSET = S65_CLK;
```

Widzimy, że po każdym ustawieniu linii DAT następuje wywołanie narastającego zbocza na linii CLK. W ten sposób wyślemy wszystkie 8 bitów argumentu `dane`.

Trzy tajemnicze linie czekają na słowo prawdy o nich: to RS, RST i BL. Za pomocą pierwszej linii sygnalizujemy wysyłanie danych (RS = 0) i rozkazów (RS = 1). Druga z przedstawionych linii posłuży nam wyłącznie do wystartowania sterownika wyświetlacza. Natomiast ustawienie linii BL włącza podświetlenie matrycy wyświetlacza S65. Jej działanie jest niezależne od zainicjowania sterownika wyświetlacza.

Skoro wiemy już, jak wysyłać dane do wyświetlacza S65, zatrzymajmy się nad jego uruchomieniem. Inicjalizacja sterownika wyświetlacza S65 składa się z 10 punktów i wymaga wysłania kilku sekwencji uruchomieniowych. Dwie z tych sekwencji radzę umieścić w osobnym pliku.

Listing dane15.h

```
typedef unsigned char int8;
typedef unsigned short int16;

//sekwencje inicjujące S65
int8 kod2[20] = {0xEF, 0x00, 0xEE, 0x04, 0x1B, 0x04, 0xFE, 0xFE, 0xFE, 0xFE,
                 0xEF, 0x90, 0x4A, 0x04, 0x7F, 0x3F, 0xEE, 0x04, 0x43, 0x06};

int8 kod3[40] = {0xEF, 0x90, 0x09, 0x83, 0x08, 0x00, 0x0B, 0xAF, 0x0A, 0x00,
         0x05, 0x00, 0x06, 0x00, 0x07, 0x00, 0xEF, 0x00, 0xEE, 0x0C,
         0xEF, 0x90, 0x00, 0x80, 0xEF, 0xB0, 0x49, 0x02, 0xEF, 0x00,
         0x7F, 0x01, 0xE1, 0x81, 0xE2, 0x02, 0xE2, 0x76, 0xE1, 0x83};
```

A oto opis procedury uruchomieniowej dla sterownika LS020xxx wyświetlacza S65:

1. Ustawienie kierunków wyjściowych linii obsługujących wyświetlacz:

   ```
   IODIR |= (S65_CS|S65_RS|S65_RST|S65_BL|S65_DAT|S65_CLK);
   ```

2. Wysłanie sekwencji rozruchowej, składającej się z 6 punktów:
   - CS = 1
   - CLK = 1
   - DAT = 1
   - RS = 1
   - RST = 0
   - RST = 1

   Sekwencję realizujemy za pomocą kodu:

   ```
   IOSET = S65_CS;
   IOSET = S65_CLK;
   IOSET = S65_DAT;
   ```

```
IOSET = S65_RS;
IOCLR = S65_RST;
IOSET = S65_RST;
```

3. Oczekiwanie 1 ms:

```
Czekaj_ms(1);
```

4. Wysłanie kodu numer 1, który składa się z 4 bajtów:

   0xFD, 0xFD, 0xFD, 0xFD

   Podpunkt procedury realizujemy za pomocą kodu:

```
Wyslij_dane_S65(0xFD);
Wyslij_dane_S65(0xFD);
Wyslij_dane_S65(0xFD);
Wyslij_dane_S65(0xFD);
```

5. Oczekiwanie 60 ms:

```
Czekaj_ms(60);
```

6. Wysłanie kodu numer 2, składającego się z 20 bajtów:

   0xEF, 0x00, 0xEE, 0x04, 0x1B, 0x04, 0xFE, 0xFE, 0xFE, 0xFE, 0xEF, 0x90, 0x4A, 0x04, 0x7F, 0x3F, 0xEE, 0x04, 0x43, 0x06

   Kod został umieszczony w pliku *dane15.h* w tablicy `kod2`. Użycie tablicy ułatwi proces programowania, gdyż do jej załadowania możemy skorzystać z pętli:

```
for(i=0; i<20; i++) Wyslij_dane_S65(kod2[i]);
```

7. Oczekiwanie 7 ms:

```
Czekaj_ms(7);
```

8. Wysłanie kodu numer 3, składającego się z 40 bajtów:

   0xEF, 0x90, 0x09, 0x83, 0x08, 0x00, 0x0B, 0xAF, 0x0A, 0x00, 0x05, 0x00, 0x06, 0x00, 0x07, 0x00, 0xEF, 0x00, 0xEE, 0x0C, 0xEF, 0x90, 0x00, 0x80, 0xEF, 0xB0, 0x49, 0x02, 0xEF, 0x00, 0x7F, 0x01, 0xE1, 0x81, 0xE2, 0x02, 0xE2, 0x76, 0xE1, 0x83

   I znów do jego wysłania użyjemy pętli:

```
for(i=0; i<40; i++) Wyslij_dane_S65(kod3[i]);
```

9. Oczekiwanie 50 ms:

```
Czekaj_ms(50);
```

10. Wysłanie kodu numer 3, składającego się z 6 bajtów:

    0x80, 0x01, 0xEF, 0x90, 0x00, 0x00

    Ponieważ kod jest krótki, do jego wysłania nie ma potrzeby używać pętli:

```
Wyslij_dane_S65(0x80);
Wyslij_dane_S65(0x01);
Wyslij_dane_S65(0xEF);
Wyslij_dane_S65(0x90);
Wyslij_dane_S65(0);
Wyslij_dane_S65(0);
```

Tak oto sterownik wyświetlacza S65 został zainicjowany. Choć nie należy to do inicjalizacji sterownika, polecam w funkcji startowej włączyć podświetlenie wyświetlacza.

```
IOSET = S65_BL;
```

Pierwsze zadanie możemy uznać za zrealizowane. Oto gotowy kod programu głównego.

Listing lekcja15_1.c

```
#include <LPC210x.H>
#include "dane15.h"

#define S65_RST 1<<16
#define S65_RS  1<<17
```

```
#define S65_CLK 1<<18
#define S65_CS  1<<19
#define S65_DAT 1<<20
#define S65_BL  1<<21

void Czekaj_ms(int c)
{
   c *= 12000;
   while(c > 0) c--;
}

void Wyslij_dane_S65(int8 dane)
{
  //sygnalizuj wysyłanie danych (poziom niski CS)
   IOCLR = S65_CS;

  //bit 7
   if(dane & 0x80) IOSET = S65_DAT; else IOCLR = S65_DAT;
   IOCLR = S65_CLK;
   IOSET = S65_CLK;

  //bit 6
   if(dane & 0x40) IOSET = S65_DAT; else IOCLR = S65_DAT;
   IOCLR = S65_CLK;
   IOSET = S65_CLK;

  //bit 5
   if(dane & 0x20) IOSET = S65_DAT; else IOCLR = S65_DAT;
   IOCLR = S65_CLK;
   IOSET = S65_CLK;

  //bit 4
   if(dane & 0x10) IOSET = S65_DAT; else IOCLR = S65_DAT;
   IOCLR = S65_CLK;
   IOSET = S65_CLK;

  //bit 3
   if(dane & 0x08) IOSET = S65_DAT; else IOCLR = S65_DAT;
   IOCLR = S65_CLK;
   IOSET = S65_CLK;

  //bit 2
   if(dane & 0x04) IOSET = S65_DAT; else IOCLR = S65_DAT;
   IOCLR = S65_CLK;
   IOSET = S65_CLK;

  //bit 1
   if(dane & 0x02) IOSET = S65_DAT; else IOCLR = S65_DAT;
   IOCLR = S65_CLK;
   IOSET = S65_CLK;

  //bit 0
   if(dane & 0x01) IOSET = S65_DAT; else IOCLR = S65_DAT;
   IOCLR = S65_CLK;
   IOSET = S65_CLK;

  //sygnalizuj koniec wysyłania danych (poziom wysoki CS)
   IOSET = S65_CS;
}

void Start_S65()
{
   int i;

//ustaw kierunek wyjściowy linii obsługujących S65
   IODIR |= (S65_CS|S65_RS|S65_RST|S65_BL|S65_DAT|S65_CLK);
```

```
//sekwencja startująca
   IOSET = S65_CS;
   IOSET = S65_CLK;
   IOSET = S65_DAT;
   IOSET = S65_RS;
   IOCLR = S65_RST;
   IOSET = S65_RST;

//zaczekaj 1 ms
   Czekaj_ms(1);

//sekwencja 1
   Wyslij_dane_S65(0xFD);
   Wyslij_dane_S65(0xFD);
   Wyslij_dane_S65(0xFD);
   Wyslij_dane_S65(0xFD);

//zaczekaj 60 ms
   Czekaj_ms(60);

//sekwencja 2
   for(i=0; i<20; i++) Wyslij_dane_S65(kod2[i]);

//zaczekaj 7 ms
   Czekaj_ms(7);

//sekwencja 3
   for(i=0; i<40; i++) Wyslij_dane_S65(kod3[i]);

//zaczekaj 50 ms
   Czekaj_ms(50);

//sekwencja 4
   Wyslij_dane_S65(0x80);
   Wyslij_dane_S65(0x01);
   Wyslij_dane_S65(0xEF);
   Wyslij_dane_S65(0x90);
   Wyslij_dane_S65(0);
   Wyslij_dane_S65(0);

//włącz podświetlenie
   IOSET = S65_BL;
}

int main()
{
  //uruchomienie wyświetlacza S65
   Start_S65();

  //pętla nieskończona
   for(;;);
}
```

Realizacja pierwszego zadania pozwoliła nam lepiej poznać tajniki sterowania wyświetlaczem S65. Zapewne przyda się także w sytuacjach, w których będą zawodziły inne próby skomunikowania się ze sterownikiem LS020xxx.

Teraz zrealizujemy to samo zadanie za pomocą interfejsu SPI. I tu na wstępie muszą pojawić się pewne uwagi natury egzystencjalnej. Otóż dokumentacja układu LPC2106 podaje, że rolą linii SSEL jest ustawianie urządzenia w tryb master lub slave. Wiadomym też jest, że w trybie master, w wyniku błędu, linia SSEL musi być zewnętrznie podciągnięta pod napięcie 3V3. Pamiętajmy o tym, gdyż niepodłączenie linii SSEL (wyprowadzenie P0.7) pod źródło napięcia jest najczęściej popełnianym błędem, który sprawia, że interfejs SPI nie działa. Pamiętajmy również, by linię podłączyć pod napięcie 3V3, a nie 5 V. W drugim przypadku mikrokontroler potrafi bardzo mocno się rozgrzać, do stanu produkowania smrodu włącznie.

Nigdy nie sprawdzałem, co dalej z gorącym mikrokontrolerem może się stać, i do takich prób nie zachęcam. Pamiętajmy: aby skorzystać z interfejsu SPI, wyprowadzenie P0.7 podłączamy pod napięcie 3V3, czyli do listwy P26 na płytce edukacyjnej.

W celu utrwalenia powyższych wskazówek radzę obejrzeć obrazki ilustrujące kilka chwil z życia Misia Ekrysia[3] (patrz rysunek 15.11).

Rysunek 15.11. *Opowieść dydaktyczna z cyklu Sekretny świat misia Ekrysia*

Zdefiniujmy nowe połączenia układu, zgodne z rysunkiem 15.8.

```
#define S65_CS  1<<10
#define S65_DAT 1<<6
#define S65_CLK 1<<4
#define S65_RS  1<<8
#define S65_RST 1<<9
#define S65_BL  1<<11
//P0.7 do 3V3!
```

Teraz zajmiemy się inicjalizacją interfejsu SPI. W tym celu musimy:

1. Wybrać alternatywną funkcję wyprowadzeń SPI. Przypomnijmy sobie tabelę 14.3. Dowiemy się z niej, że do wyprowadzeń SPI należą linie P0.4 (SCK), P0.5 (MISO), P0.6 (MOSI) oraz P0.7 (SSEL). Aby pełniły rolę wyprowadzeń SPI, należy bity rejestru PINSEL0 od 8 do 15 ustawić wartościami 01. Otrzymamy liczbę binarną 0b00000000000000000101010100000000, której odpowiada liczba szesnastkowa 0x00005500. Otrzymujemy zatem instrukcję:

   ```
   PINSEL0 = 0x00005500;
   ```

[3] Geneza powstania imienia Ekryś owiana jest mgłą tajemnicy. Za jego twórcę uchodzi legendarny badacz natury zwierząt Vasco da Vasco. Miałby jakoby ów wielki podróżnik, spotkawszy przodka Ekrysia, wypowiedzieć dwa znamienne słowa, co prawda naprędce, może dlatego nieco niedbale, które w pewnej parafrazie przetrwały do dziś. Słowa te podobno brzmiały: „Nie gryź!”.

2. Wybrać prędkość pracy interfejsu SPI. Robi się to poprzez przypisanie dzielnika wielkości PCLK/SPCCR do rejestru S0SPCCR. Dokumentacja podaje, że minimalną wielkością może być 8. Z tego wynika, że SPI może pracować z maksymalną prędkością 60 000 000/8 = 7 500 000 Hz:

```
S0SPCCR = 8;
```

3. Ustawić tryb pracy SPI. Służy do tego rejestr SPCR. Najważniejsze bity tego rejestru to:
   - CPOL — bit 4. Służy ustawieniu polaryzacji potwierdzania danych linią SCK. Wartość 0 oznacza potwierdzanie zboczem narastającym, wartość 1 oznacza potwierdzanie zboczem opadającym. Pamiętamy sposób wysyłania danych do sterownika wyświetlacza S65 i domyślamy się, że dla naszych zastosowań bit CPOL powinien być wyzerowany.
   - MSTR — bit 5. Służy konfiguracji pracy SPI jako master (bit = 1) lub slave (bit = 0). W naszym programie bit MSTR musi być ustawiony (tryb master).
   - LSBF — kierunek wysyłania bitów danych. Jeśli LSBF = 1, dane są wysyłane w trybie LSB (bit 0 pierwszy), jeśli LSBF = 0 dane są wysyłane w trybie MSB (bit 7 pierwszy). Zgodnie z podanym opisem wysyłania danych do sterownika wyświetlacza S65 dane mają być wysyłane w trybie MSB, czyli bit LSBF powinien być wyzerowany.

   Ostatecznie otrzymujemy takie przypisanie:

```
S0SPCR  = 0x20;
```

Funkcja inicjalizacji SPI została skonstruowana. Zobaczmy ją w całości:

```
void Start_SPI()
{
   PINSEL0 = 0x00005500;
   S0SPCCR = 8;
   S0SPCR  = 0x20;//(transmisja 8-bitowa MSB, tryb Master)
}
```

Wysyłanie danych zaimplementujemy w funkcji o następującej deklaracji:

```
void Wyslij_dane(int8 dane);
```

Pamiętamy, że wysyłanie danych do wyświetlacza powinno odbywać się przy wyzerowanej linii CS. Mamy na razie szkielet funkcji `Wyslij_dane`.

```
void Wyslij_dane(int8 dane)
{
  //sygnalizuj wysyłanie danych (poziom niski CS)
   IOCLR = S65_CS;

  ////////////////////////////////////////
  //tu będą instrukcje transmisji danych
  ////////////////////////////////////////

  //sygnalizuj koniec wysyłania danych (poziom wysoki CS)
   IOSET = S65_CS;
}
```

Wysłanie danych poprzez interfejs SPI wiąże się z użyciem jedynie dwu instrukcji. Pierwsza z nich to załadowanie do rejestru S0SPDR bajtu do wysłania.

```
S0SPDR = dane;
```

W drugiej instrukcji oczekujemy na potwierdzenie wysłania danych. Zakończenie wysyłania danych jest sygnalizowane ustawieniem bitu SPIF rejestru SPSR. Bit SPIF jest siódmym bitem rejestru SPSR, dlatego druga instrukcja wysyłania danych ma następującą postać:

```
while((S0SPSR&0x80) == 0);
```

W ten sposób zrealizowaliśmy drugie zadanie w ramach tej lekcji.

Listing lekcja15_2.c

```
#include <LPC210x.h>
#include "dane15.h"

#define S65_CS  1<<10
#define S65_DAT 1<<6
#define S65_CLK 1<<4
#define S65_RS  1<<8
#define S65_RST 1<<9
#define S65_BL  1<<11
//P0.7 do 3V3!

void Czekaj_ms(int c)
{
   c *= 12000;
   while(c > 0) c--;
}

void Start_SPI()
{
   PINSEL0 = 0x00005500;
   S0SPCCR = 8;
   S0SPCR  = 0x20;//(transmisja 8-bitowa MSB, tryb Master)
}

void Wyslij_dane(int8 dane)
{
  //sygnalizuj wysyłanie danych (poziom niski CS)
   IOCLR = S65_CS;

  //załaduj dane
   S0SPDR = dane;

  //zaczekaj na potwierdzenie wysłania danych
   while((S0SPSR&0x80) == 0);

  //sygnalizuj koniec wysyłania danych (poziom wysoki CS)
   IOSET = S65_CS;
}

void Start_S65()
{
   int i;

//ustaw kierunek wyjściowy linii obsługujących S65
   IODIR |= (S65_CS|S65_RS|S65_RST|S65_BL);

//sekwencja startująca
   IOSET = S65_CS;
   IOSET = S65_CLK;
   IOSET = S65_DAT;
   IOSET = S65_RS;
   IOCLR = S65_RST;
   IOSET = S65_RST;

//zaczekaj 1 ms
   Czekaj_ms(1);

//sekwencja 1
   Wyslij_dane(0xFD);
   Wyslij_dane(0xFD);
   Wyslij_dane(0xFD);
   Wyslij_dane(0xFD);
```

```
//zaczekaj 60 ms
   Czekaj_ms(60);

//sekwencja 2
   for(i=0; i<20; i++) Wyslij_dane(kod2[i]);

//zaczekaj 7 ms
   Czekaj_ms(7);

//sekwencja 3
   for(i=0; i<40; i++) Wyslij_dane(kod3[i]);

//zaczekaj 50 ms
   Czekaj_ms(50);

//sekwencja 4
   Wyslij_dane(0x80);
   Wyslij_dane(0x01);
   Wyslij_dane(0xEF);
   Wyslij_dane(0x90);
   Wyslij_dane(0);

//włącz podświetlenie
   IOSET = S65_BL;
}

int main()
{
  //start modułu SPI
   Start_SPI();

  //włączenie wyświetlacza
   Start_S65();

  //nieskończona pętla
   for(;;);
}
```

Dwa poprzednie zadania można nazwać rozruchowymi. Czekają nas programy coraz większe, z coraz liczniejszą grupą funkcji. Zasadne więc wydaje się podzielenie programu na kilka plików. Naszym celem jest zbudowanie biblioteki obsługi wyświetlacza S65. Kolejne etapy na drodze do tego świątłego celu oznaczać będziemy numerami. Natomiast produkt końcowy, czyli pliki nagłówkowe wspaniałej biblioteki funkcji obsługi wyświetlacza S65, nazwiemy po prostu *dane.h* oraz *S65.h*.

Jeden z plików nagłówkowych możemy podać już teraz, z nazwą sugerującą ostateczne rozwiązanie. To plik zawierający funkcję opóźniającą.

Listing czekaj.h

```
void Czekaj_ms(int c)
{
   c *= 12000;
   while(c > 0) c--;
}
```

Plik nagłówkowy zawierający sekwencje startowe sterownika wyświetlacza S65 uzupełnimy o sekwencję uruchamiającą rozkaz wypełnienia tła kolorem.

Listing dane15_3.h

```
typedef unsigned char int8;
typedef unsigned short int16;

//sekwencje inicjujące S65
int8 kod2[20] = {0xEF, 0x00, 0xEE, 0x04, 0x1B, 0x04, 0xFE, 0xFE, 0xFE, 0xFE,
                 0xEF, 0x90, 0x4A, 0x04, 0x7F, 0x3F, 0xEE, 0x04, 0x43, 0x06};
```

```
int8 kod3[40] = {0xEF, 0x90, 0x09, 0x83, 0x08, 0x00, 0x0B, 0xAF, 0x0A, 0x00,
                 0x05, 0x00, 0x06, 0x00, 0x07, 0x00, 0xEF, 0x00, 0xEE, 0x0C,
         0xEF, 0x90, 0x00, 0x80, 0xEF, 0xB0, 0x49, 0x02, 0xEF, 0x00,
         0x7F, 0x01, 0xE1, 0x81, 0xE2, 0x02, 0xE2, 0x76, 0xE1, 0x83};

//sekwencja kodu wypełnienia
int8 kod_wypelnienia[8] = {0xEF, 0x90, 0x05, 0x04, 0x06, 0x00, 0x07, 0x00};
```

Funkcje obsługi wyświetlacza, także funkcję inicjującą interfejs SPI, umieścimy w pliku *S65_15_3.h*. Numerowana nazwa sugeruje, że nie jest to gotowa biblioteka funkcji obsługi wyświetlacza. W pliku, prócz znanych funkcji, znajdą się dwie nowe i na nich się teraz skoncentrujmy.

Pierwsza z nowych funkcji zapełni tło matrycy wyświetlacza jednolitym kolorem. Pierwsze zdanie i już nieprawda — kolor wcale nie musi być jednolity. Ale o tym za chwilę. Funkcja będzie się nazywała `Rysuj_tlo_S65` i będzie pobierała 3 argumenty — składowe koloru tła:

```
void Rysuj_tlo_S65(int16 R, int16 G, int16 B);
```

Na wyświetlaczu S65 kolor jest kodowany za pomocą 16-bitowej wartości, zawierającej składowe koloru czerwonego, zielonego i niebieskiego. Pięć najstarszych bitów zawiera składową koloru czerwonego, bity od piątego do dziesiątego zawierają składową zieloną, pięć najmłodszych bitów zawiera składową koloru niebieskiego (patrz rysunek 15.12).

Rysunek 15.12. *Kodowanie koloru w standardzie TFT*

Przedstawiony sposób kodowania koloru nazywa się kodowaniem w standardzie TFT. Pamiętamy, że wysyłamy do wyświetlacza dane bajtowej wielkości[4]. Musimy więc na przekazanych do funkcji argumentach składowych koloru wykonać dwie operacje umożliwiające ich wysłanie. Po pierwsze z trzech wartości, dwóch pięcio- i jednej sześciobitowej, musimy utworzyć jedną wartość 16-bitową. Na przykład tak:

```
int16 i;
int8 kolor1, kolor0;

//ustaw kolor
i = (R<<11)|(G<<5)|(B);
```

Teraz z wartości 16-bitowej musimy utworzyć dwie wielkości 8-bitowe.

```
kolor0 = i;
kolor1 = i>>8;
```

Kolor jest przygotowany do wysłania. Zanim zajmiemy się jego ładowaniem do pamięci wyświetlacza, musimy wysłać sekwencję bajtów oznaczającą rozkaz wypełnienia tła zadanym kolorem. Ponieważ wysyłamy rozkaz, linia RS musi być w wysokim stanie logicznym.

```
IOSET = S65_RS;
```

Sekwencja rozkazu jest umieszczona w pliku nagłówkowym *dane15_3.h*. Dzięki załadowaniu jej do tablicy możliwe staje się użycie pętli:

```
for(i=0; i<8; i++) Wyslij_dane(kod_wypelnienia[i]);
```

Rozkaz wydany, czas na wysłanie koloru. Ponieważ teraz będziemy wysyłać dane *sensu stricte*, linia RS musi być wyzerowana.

```
IOCLR = S65_RS;
```

[4] Nic nie stoi na przeszkodzie, by do wyświetlacza S65 wysyłać dane wielkości 16-bitowej. Nie czynimy tego, gdyż mikrokontroler LPC2106 posiada wyłącznie 8-bitowy interfejs SPI. Można natomiast spróbować wysyłać paczki większych rozmiarów na przykład za pomocą mikrokontrolera LPC2103.

Rozdzielczość matrycy 132×176 pikseli daje w sumie konieczność załadowania 23 232 pikseli określonym kolorem. Nie ma sprawy. Zróbmy to.

```
for(i=0; i<23232; i++)
{
   Wyslij_dane(kolor1);
   Wyslij_dane(kolor0);
}
```

To wszystko. Tło jest namalowane. Postać pętli nasuwa nam od razu myśl, że kolor tła nie musi być jednolity. Rzeczywiście. Wystarczy zmienić zapis na taki:

```
for(i=0; i<23232; i++)
{
   Wyslij_dane(kolor1++);
   Wyslij_dane(kolor0);
}
```

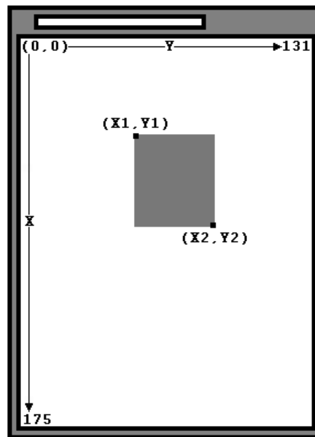
a otrzymamy tło pokryte ładnie mieniącymi się przejściami z koloru zielonego w czerwień. Można także ładować dane z tablicy i w ten sposób wyświetlać obrazy.

Druga z nowych funkcji jest bardzo podobna do pierwszej. Służy do zapełnienia fragmentu obszaru matrycy kolorem czy też kolorami. A ponieważ jest to jedyna funkcja tego typu dostępna na wyświetlaczu S65, czyni ją to podstawową funkcją rysującą wyświetlacza S65.

Deklaracja funkcji jest dość rozbudowana.

```
void Rysuj_obszar_S65(int8 X1, int8 Y1,
                      int8 X2, int8 Y2,
            int16 R, int16 G, int16 B);
```

Parametry X1 i Y1 oznaczają lewy górny róg pola do zamalowania, parametry X2 i Y2 definiują prawy dolny róg tego obszaru. Wszelkie wątpliwości powinien rozwiać rysunek 15.13.



Rysunek 15.13. *Współrzędne ekranu wyświetlacza i znaczenie czterech pierwszych parametrów użytych w deklaracji funkcji rysującej obszar*

Trzy ostatnie parametry funkcji `Rysuj_obszar_S65` to oczywiście składowe koloru. W funkcji zakładamy, że do obszaru należą także punkty z szerokości `Y2` i wysokości `X2`. Dlatego wielkość obszaru do zamalowania obliczamy za pomocą wzoru:

```
obszar = (X2-X1+1)*(Y2-Y1+1);
```

Teraz powinna nastąpić sekwencja rozkazu zamalowania obszaru. Do tej sekwencji wplatamy przekazane przez parametry współrzędne obszaru do odmalowania. Zwróćmy uwagę na wiersze 6., 8., 10. i 12.

```
Wyslij_dane(0xEF);
Wyslij_dane(0x90);
Wyslij_dane(0x05);
Wyslij_dane(0x00);
```

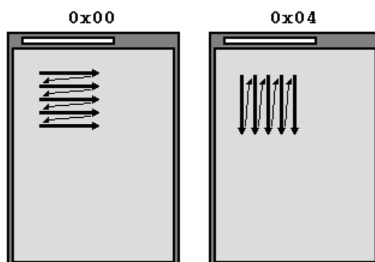
```
Wyslij_dane(0x08);
Wyslij_dane(Y1);
Wyslij_dane(0x09);
Wyslij_dane(Y2);
Wyslij_dane(0x0A);
Wyslij_dane(X1);
Wyslij_dane(0x0B);
Wyslij_dane(X2);
```

Ponieważ wysyłamy rozkaz, musimy pamiętać, by poinformować o tej strasznej prawdzie sterownik wyświetlacza, a to za pomocą ustawienia wysokiej wartości linii RS.

Przyjrzyjmy się wysyłanej sekwencji:

0xEF, 0x90, 0x05, 0x00, 0x08, Y1, 0x09, Y2, 0x0A, X1, 0x0B, X2

Jest to zbiór liczb i parametrów. Co może być w nim ciekawego? A jednak coś może być. Zwróćmy uwagę na bajt czwarty. Ma on wartość 0x00. Jest to bajt kierunku rysowania, nazwijmy go bajtem kierunkowym. Jego możliwe wartości i związany z tym kierunek rysowania obszarów prezentuje rysunek 15.14.



Rysunek 15.14. *Ilustracja wpływu wartości bajtu kierunkowego na kierunek rysowania obszarów*

Dzięki zmianie kierunku rysowania obszarów możemy wybrać wygodniejszą dla nas orientację wyświetlacza: pionową, jak dotychczas, lub poziomą.

Ostatnim zadaniem funkcji `Rysuj_obszar_S65` powinno być wypełnienie obszaru określonym kolorem. W ten sposób mamy gotowy plik *S65_15_3.h*.

Listing S65_15_3.h

```
#define S65_CS  1<<10
#define S65_DAT 1<<6
#define S65_CLK 1<<4
#define S65_RS  1<<8
#define S65_RST 1<<9
#define S65_BL  1<<11
//Uwaga! Linia P0.7 do 3V3!

void Start_SPI()
{
   PINSEL0 = 0x00005500;
   S0SPCCR = 8;
   S0SPCR  = 0x20;//(transmisja 8-bitowa, tryb Master)
}

void Wyslij_dane(int8 dane)
{
  //sygnalizuj wysyłanie danych (poziom niski CS)
   IOCLR = S65_CS;

  //załaduj dane
   S0SPDR = dane;
```

```
  //zaczekaj na potwierdzenie wysłania danych
    while((S0SPSR&0x80) == 0);

  //sygnalizuj koniec wysyłania danych (poziom wysoki CS)
    IOSET = S65_CS;
}

void Start_S65()
{
    int i;
//włącz interfejs SPI
    Start_SPI();

//ustaw kierunek wyjściowy linii obsługujących S65
    IODIR |= (S65_CS|S65_RS|S65_RST|S65_BL);

//sekwencja startująca
    IOSET = S65_CS;
    IOSET = S65_CLK;
    IOSET = S65_DAT;
    IOSET = S65_RS;
    IOCLR = S65_RST;
    IOSET = S65_RST;

//zaczekaj 1 ms
    Czekaj_ms(1);

//sekwencja 1
    Wyslij_dane(0xFD);
    Wyslij_dane(0xFD);
    Wyslij_dane(0xFD);
    Wyslij_dane(0xFD);

//zaczekaj 60 ms
    Czekaj_ms(60);

//sekwencja 2
    for(i=0; i<20; i++) Wyslij_dane(kod2[i]);

//zaczekaj 7 ms
    Czekaj_ms(7);

//sekwencja 3
    for(i=0; i<40; i++) Wyslij_dane(kod3[i]);

//zaczekaj 50 ms
    Czekaj_ms(50);

//sekwencja 4
    Wyslij_dane(0x80);
    Wyslij_dane(0x01);
    Wyslij_dane(0xEF);
    Wyslij_dane(0x90);
    Wyslij_dane(0);

//włącz podświetlenie
    IOSET = S65_BL;
}

void Rysuj_tlo_S65(int16 R, int16 G, int16 B)
{
    int16 i;
    int8 kolor1, kolor0;
```

```
  //ustaw kolor
   i = (R<<11)|(G<<5)|(B);
   kolor0 = i;
   kolor1 = i>>8;

  //sygnalizuj rozkaz (RS = 1)
   IOSET = S65_RS;

  //załaduj kod wypełniania obszaru
   for(i=0; i<8; i++) Wyslij_dane(kod_wypelnienia[i]);

  //sygnalizuj dane (RS = 0)
   IOCLR = S65_RS;

  //wypełnij kolorem
   for(i=0; i<23232; i++)
   {
      Wyslij_dane(kolor1);
      Wyslij_dane(kolor0);
   }
}

void Rysuj_obszar_S65(int8 X1, int8 Y1,
                      int8 X2, int8 Y2,
               int16 R, int16 G, int16 B)
{
   int obszar;
   int8 kolor1, kolor0;

  //ustaw kolor
   obszar = (R<<11)|(G<<5)|(B);
   kolor0 = obszar;
   kolor1 = obszar>>8;

  //ustaw wielkość pola
   obszar = (X2-X1+1)*(Y2-Y1+1);

  //sygnalizuj rozkaz (RS = 1)
   IOSET = S65_RS;

  //sekwencja uruchamiająca rysowanie obszaru
   Wyslij_dane(0xEF);
   Wyslij_dane(0x90);
   Wyslij_dane(0x05);
   Wyslij_dane(0x00);
   Wyslij_dane(0x08);
   Wyslij_dane(Y1);
   Wyslij_dane(0x09);
   Wyslij_dane(Y2);
   Wyslij_dane(0x0A);
   Wyslij_dane(X1);
   Wyslij_dane(0x0B);
   Wyslij_dane(X2);

  //sygnalizuj dane (RS = 0)
   IOCLR = S65_RS;

  //wypełnij kolorem
   for(; obszar>0; obszar--)
   {
      Wyslij_dane(kolor1);
      Wyslij_dane(kolor0);
   }
}
```

Pozostało nam skorzystać z zasobów pliku *S65_15_3.h* i zapełnić matrycę wyświetlacza pożądanymi figurami. Zrobimy to w funkcji `main` głównego pliku projektu, noszącego nazwę *lekcja15_3.c*. Musimy pamiętać, by najpierw dodać zdefiniowane wcześniej pliki nagłówkowe do zasobów projektu. Wystarczy, że zasoby dodamy do pliku głównego za pomocą dyrektywy `include` — w ten sposób automatycznie zostaną dodane do zasobów projektu.

```
#include <LPC210x.H>
#include "czekaj.h"
#include "dane15_3.h"
#include "S65_15_3.h"
```

Być może bystry obserwator zauważył, że funkcję inicjującą interfejs SPI wywołujemy w funkcji `Start_S65`. To oznacza, że nie musimy jej umieszczać w funkcji `main`, wystarczy przecież wywołanie funkcji `Start_S65`.

Po wystartowaniu wyświetlacza S65 na matrycy widzimy różnokolorowy szum. Aby się go pozbyć, wypełnimy tło jednolitym kolorem, na przykład zielonym.

```
Rysuj_tlo_S65(0, 0x3F, 0);
```

Pamiętamy, że parametry funkcji odpowiadają składowym koloru czerwonego, zielonego i niebieskiego. Kolor zielony jest kodowany za pomocą sześciu bitów. Ustawienie wszystkich bitów tworzy liczbę 0x3F. Następnie rysujemy dwa niebieskie kwadraty.

```
   Rysuj_obszar_S65(20, 20, 40, 40, 0, 0, 0x1F);
   Rysuj_obszar_S65(20, 91, 40, 111, 0, 0, 0x1F);
```

Niestety przy programowaniu sterownika wyświetlacza S65 nie możemy wydać rozkazu w stylu *narysuj trójkąt*. Tę i wiele podobnych figur musimy skonstruować samodzielnie z prostokątnych obszarów. Jeśli w funkcji `Rysuj_obszar_S65` zajdzie przypadek, w którym X1 = X2, otrzymamy poziomą linię. Linię pionową narysujemy, gdy Y1 = Y2. Trójkąt narysujemy z coraz dłuższych poziomych linii.

```
for(k=0; k<40; k++)
      Rysuj_obszar_S65(50+k, 66-k, 50+k, 66+k, 0x1F, 0, 0);
```

Mówimy o kwadratach, trójkątach, a przecież szkicujemy twarz pana Ziutka. Powinniśmy więc mówić o oczach, nosie i zębach, które musimy jeszcze narysować. Samo naszkicowanie obszarów będzie łatwe. Natomiast podstawowe kolory osiągniemy w efekcie przesunięcia wartości początkowej 0x1F w każdym przebiegu pętli o krok w lewo. Popatrzmy na sześć pierwszych kroków operacji przesuwania w lewo wartości 16-bitowej:

```
0b0000000000011111 = 0x001F
0b0000000000111110 = 0x003E
0b0000000001111100 = 0x007C
0b0000000011111000 = 0x00F8
0b0000000111110000 = 0x01F0
0b0000001111100000 = 0x03E0
```

Widzimy, że z kodu koloru niebieskiego otrzymaliśmy kod koloru zielonego. Wreszcie cały program jest gotowy.

Listing lekcja15_3.c

```
#include <LPC210x.H>
#include "czekaj.h"
#include "dane15_3.h"
#include "S65_15_3.h"

int main()
{
   unsigned int kolor = 0x1F, k;

  //uruchomienie wyświetlacza S65
   Start_S65();

  //tło w kolorze zielonym
   Rysuj_tlo_S65(0, 0x3F, 0);
```

```
  //narysuj dwa niebieskie kwadraty
    Rysuj_obszar_S65(20, 20, 40, 40, 0, 0, 0x1F);
    Rysuj_obszar_S65(20, 91, 40, 111, 0, 0, 0x1F);

  //narysuj czerwony trójkąt
    for(k=0; k<40; k++)
       Rysuj_obszar_S65(50+k, 66-k, 50+k, 66+k, 0x1F, 0, 0);

  //narysuj czarny prostokąt
    Rysuj_obszar_S65(102, 7, 158, 123, 0, 0, 0);

  //narysuj obszary w przekrojowych kolorach
    for(k=10; k<117; k+=7)
    {
       Rysuj_obszar_S65(105, k, 155, k+5,
             (unsigned char)((kolor>>11)&(0x1F)),
             (unsigned char)((kolor>>5)&(0x3F)),
             (unsigned char)(kolor&0x1F));
       kolor = (kolor<<1);
    }

  //pętla nieskończona
    for(;;);
}
```

15.2. Asembler

Zagadnienia:

- ♦ **Kłopoty doskonałego programisty asemblerowego.**
- ♦ **Sposób przekazywania parametrów do podprogramów.**
- ♦ **Podprogram rysowania obszarów.**

Szanowny Czytelniku (właściwie chciałem powiedzieć: drogi ekspercie w programowaniu układów ARM7 w asemblerze), poznałeś już większość tajemnic programowania niskopoziomowego układów ARM7, co pozwala Ci pisać dowolnie skomplikowany program. To, jak również fakt, że będziemy zajmowali się programami coraz dłuższymi, sprawia, że nie będę już w paragrafach asemblerowych zamieszczał kompletnych listingów budowanych programów. Zamiast tego będziemy omawiać wybrane ciekawsze fragmenty konstruowanego kodu[5]. Po prostu nie chcę, aby niniejszy podręcznik swoją objętością zaczął dorównywać dziełom Lenina.

Paragraf zaczniemy od kilku uwag ogólnych. Otóż młodych programistów asemblera często nurtują następujące pytania:

1. W jaki sposób argumenty powinny być przekazywane do podprogramów? Czy należy użyć rejestrów, czy odkładać dane na stos, czy może umieszczać je pod specjalnym adresem pamięci RAM?
2. W jakiego typu kod niskopoziomowy zamieniają kompilatory nasz kod wysokiego poziomu? (To właściwie druga postać pierwszego pytania).
3. W jaki sposób wartości są zwracane z podprogramów?

Należy wiedzieć, że nie zawsze sposób radzenia sobie z parametrami przez kompilatory jest optymalny. Więc odpowiedź na pytanie drugie wcale nie musi być tożsama z odpowiedzią na pytanie pierwsze. Pewne praktyki są jednak wspólne, przecież kompilatory też zaprojektowali programiści.

Najlepszą i najczęściej stosowaną praktyką, o ile program na to pozwala, jest użycie rejestrów. Wiadomo, że w efekcie pracy z rejestrami powstaje najszybszy kod. Zarówno argumenty podprogramów (funkcji), jak i wartości zwracane przez funkcje powinny być umieszczane w rejestrach. Dopiero w przypadku gdy danych

[5] Oczywiście pełne wersje programów znajdzie Czytelnik na dołączonym do książki nośniku CD.

jest bardzo dużo, można skorzystać ze stosu lub z innego fragmentu pamięci RAM. Dissasemblowanie programów napisanych w językach wysokiego poziomu pozwala zauważyć pewne mechanizmy wspólnie występujące we wszystkich kompilatorach. Na przykład wartość zwracana z funkcji jest najczęściej umieszczana w najmłodszym rejestrze ogólnego użytku. W asemblerze PC jest to rejestr AX (EAX), w asemblerze AVR jest to rejestr R16, wreszcie w asemblerze ARM7 jest to rejestr R0.

Jako się rzekło — kod tworzony przez kompilatory nie zawsze jest optymalny. Prześledźmy następujący przykład — przy założeniu, że w jednym z naszych programów wystąpił taki kod:

```
int fun(int a, int b)
{
   return a+b;
}

int main()
{
   int x = 2, y = 3;

   x = fun(x, y);

   for(;;);
}
```

Przymknijmy oko na to, że zaprezentowany kod jest zupełnie bezsensowny. Interesuje nas, w jaki sposób kompilator zakoduje przekazanie parametrów `x` i `y` do funkcji `fun`, a także w jaki sposób obliczona wartość zostanie zwrócona do zmiennej `x`. Program został skompilowany kompilatorem uVision3. Po jego zdissasemblowaniu zobaczymy takie dziwy:

```
0x00000210  E1A02000  MOV      R2,R0
;    5:        return a+b;
0x00000214  E0820001  ADD      R0,R2,R1
;    6: }
;    7:
;    8: int main()
0x00000218  E12FFF1E  BX       R14
;    9: {
0x0000021C  E92D4010  STMDB    R13!,{R4,R14}
;   10:        int x = 2, y = 3;
;   11:
0x00000220  E3A03002  MOV      R3,#0x00000002
0x00000224  E3A04003  MOV      R4,#0x00000003
;   12:        x = fun(x, y);
;   13:
0x00000228  E1A01004  MOV      R1,R4
0x0000022C  E1A00003  MOV      R0,R3
0x00000230  EBFFFFF6  BL       fun(0x00000210)
;   14:        for(;;);
0x00000234  E1A00000  NOP
0x00000238  EAFFFFFE  B        0x00000238
```

Widzimy, że zmienna `x` została umiejscowiona w rejestrze R3, zmienna `y` w rejestrze R4.

```
0x00000220  E3A03002  MOV      R3,#0x00000002
0x00000224  E3A04003  MOV      R4,#0x00000003
```

Zgodnie ze standardem języka C do funkcji `fun` powinny zostać przekazane kopie zmiennych. Tak się rzeczywiście stało. Wartość rejestru R4 została skopiowana do rejestru R1, wartość rejestru R3 została skopiowana do rejestru R0.

```
0x00000228  E1A01004  MOV      R1,R4
0x0000022C  E1A00003  MOV      R0,R3
```

Teraz zachodzi wywołanie funkcji `fun`. Po skopiowaniu wartości z R0 do R2 następuje zwrócenie obliczonej sumy do rejestru R0.

```
0x00000210  E1A02000  MOV      R2,R0
;    5:        return a+b;
0x00000214  E0820001  ADD      R0,R2,R1
```

A w jaki sposób przedstawiony fragment kodu wysokiego poziomu zapisalibyśmy my, doskonali programiści? Po pierwsze widać, że kopiowanie wartości rejestrów nie jest potrzebne, gdyż w funkcji `fun` nie występują działania modyfikujące wartości zmiennych, a dokładniej mówiąc — rejestrów. Po drugie w ogóle nie użylibyśmy funkcji `fun`, gdyż działanie całego przedstawionego programu sprowadza się do wykonania jednej instrukcji — sumowania. Tak kod niskopoziomowy napisalibyśmy my — doskonali programiści:

```
;int x = 2, y = 3;
MOV R3,#0x00000002
MOV R4,#0x00000003
;x = fun(x, y);
ADD R3,R4,R3
```

Wróćmy do pierwszego zadania, choć nie rozstajemy się jeszcze ze światem programistycznych rozterek. Na początek zdefiniujemy połączenie wyświetlacza S65 z układem LPC2106, tak jak prezentuje to rysunek 15.6.

```
S65_RST EQU (1<<16)
S65_RS  EQU (1<<17)
S65_CLK EQU (1<<18)
S65_CS  EQU (1<<19)
S65_DAT EQU (1<<20)
S65_BL  EQU (1<<21)
```

W programie realizującym zadanie do przekazywania parametrów do podprogramów używam rejestrów. Aby się nie pogubić w meandrach zarezerwowanych rejestrów, na początku programu umieszczam komentarz informujący mnie o stanie ich wykorzystania.

```
; R0 — rejestr ogólnego użytku
; R1 — adres IOSET
; R2 — adres IOCLR
; R3 — rejestr ogólnego użytku i używany w procedurach czekaj
; R4 — liczba µs i ms w procedurach czekaj
;    — kod do wysłania do S65
```

Z podanego opisu możemy wyczytać, że rejestr R4 jest używany do przekazywania argumentu do podprogramu oczekującego oraz do podprogramu wysyłającego dane do wyświetlacza. Zajmijmy się drugim z wymienionych podprogramów. Niech się nazywa `Wyslij_dane`. Wiadomo, że transfer danych powinien być zasygnalizowany niskim stanem linii CS.

```
Wyslij_dane
   ;sygnalizuj wysyłanie danych (poziom niski CS)
      LDR R0, =S65_CS
      STR R0, [R2]
```

W rejestrach R0 i R3 zapamiętamy też numery bitów odpowiadające liniom DAT i CLK.

```
      LDR R0, =S65_DAT
      LDR R3, =S65_CLK
```

A teraz przejdźmy do operacji testowania bitów rejestru R4. Dlaczego R4? Jak już wspomnieliśmy, za pośrednictwem tego rejestru są bowiem przenoszone dane do podprogramu `Wyslij_dane`. Popatrzmy na sposób testowania bitów 7 i 6:

```
   ;testuj 7 bit
      TST R4, #0x80
   ;jeśli bit ustawiony IOSET = S65_DAT
      STRNE R0, [R1]
   ;jeśli bit wyzerowany IOCLR = S65_DAT
      STREQ R0, [R2]
   ;IOCLR = S65_CLK
      STR R3, [R2]
   ;IOSET = S65_CLK
      STR R3, [R1]

   ;testuj  bit 6
      TST R4, #0x40
   ;jeśli bit ustawiony IOSET = S65_DAT
      STRNE R0, [R1]
   ;jeśli bit wyzerowany IOCLR = S65_DAT
```

```
        STREQ R0, [R2]
    ;IOCLR = S65_CLK
        STR R3, [R2]
    ;IOSET = S65_CLK
        STR R3, [R1]
```

Skorzystaliśmy z umiejętności dodawania do instrukcji mnemoników warunkowych. Jeśli instrukcja TST wykryje ustawiony bit rejestru R4, wykona się instrukcja STRNE, gdyż mnemonik NE aktywuje instrukcję, gdy w wyniku operacji arytmetycznej otrzymano wartość różną od 0. Nie wykona się za to instrukcja STREQ, która stanie się aktywna, gdy w wyniku operacji TST otrzymamy wartość 0. Po ustawieniu linii DAT potwierdzenie wysłania ważnych danych realizujemy narastającym zboczem CLK.

Tyle ciekawostek o realizacji pierwszego zadania w ramach tej lekcji. W drugim zadaniu wykorzystamy interfejs SPI. W związku z drugim zadaniem warto omówić postać podprogramu wysyłającego dane do wyświetlacza. A wygląda on tak:

```
;wyślij dane z pierwszych 8 bitów R4 do S65
Wyslij_dane
    ;sygnalizuj wysyłanie danych (poziom niski CS)
        LDR R0, =S65_CS
        STR R0, [R2]

      ;wysłanie kodu
    ;S0SPDR = dane
        LDR R0, =0xE0020008
        STRB R4, [R0]

      ;zaczekaj na potwierdzenie wysłania danych
    ;while((S0SPSR&0x80) == 0)
        LDR R3, =0xE0020004
petla_Wyslij_dane
        LDR R0, [R3]
        TST R0, #0x00000080
        BEQ petla_Wyslij_dane

      ;sygnalizuj koniec wysyłania danych (poziom wysoki CS)
        LDR R0, =S65_CS
        STR R0, [R1]

        ;powrót z podprogramu
        BX LR
```

Dane do wysłania są przenoszone za pośrednictwem rejestru R4. Oczywiście wysłanie danych musi być poprzedzone wyzerowaniem linii CS. Następnie bajt z rejestru R4 jest ładowany do rejestru S0SPDR o adresie 0xE0020008. Skopiowanie bajtu realizuje instrukcja STRB. W pętli oczekującej na potwierdzenie wysłania danych testujemy bit SPIF, wykonując pętlę, dopóki jest on wyzerowany.

Przystępujemy do realizacji trzeciego zadania. Zdefiniujemy podprogramy zapełniające obszary kolorem. Będą to:

- Rysuj_tlo — podprogram zapełniający jednolitym kolorem tło.
- Rysuj_obszar — podprogram zamalowujący jednolitym kolorem określony obszar.

Od razu nasuwa się pytanie o sposób przenoszenia parametrów do podprogramu. Proponuję użyć rejestru R4 do przenoszenia parametru koloru, rejestru R5 do przenoszenia współrzędnych zamalowywanego obszaru.

Kod koloru będzie umieszczany w młodszych 16 bitach rejestru R4. Rezygnujemy tym samym z ułatwienia, którego używaliśmy w języku C, pozwalającego umieszczać składowe koloru w osobnych parametrach R, G i B. Upraszczamy sposób przenoszenia parametrów do podprogramu, gdyż zależy nam na otrzymaniu jak najszybszego kodu. Popatrzmy na asemblerową postać procedury Rysuj_tlo:

```
Rysuj_tlo
    ;zapamiętaj na stosie adres powrotu z procedury
        STR R14,[R13,#-0x0004]!
```

```
    ;zapamiętaj na stosie kolor
       STR R4,[R13,#-0x0004]!

    ;sygnalizuj rozkaz (RS = 1)
       LDR R0, =S65_RS
       STR R0, [R1]

    ;wypełnij obszar wyświetlacza
    ;załaduj rozmiar tablicy
       LDR R6, =8
    ;załaduj adres pierwszego bajtu tablicy
       LDR R5, =kod_wypelnienia
petla_Rysuj_tlo_1
    ;załaduj bajt spod adresu R5 i zwiększ adres o 1
       LDRB R4,[R5],#1
       BL Wyslij_dane
       SUBS R6, R6, #1
    ;wróć, jeśli nie ma zera
       BNE petla_Rysuj_tlo_1

    ;sygnalizuj dane (RS = 0)
       LDR R0, =S65_RS
       STR R0, [R2]

    ;odzyskaj ze stosu kolor
       LDR R4,[R13],#0x0004

    ;wypełnij kolorem
       LDR R6, =23232    ;132*176
petla_Rysuj_tlo_2
    ;załaduj 2 bajty koloru
       ROR R4, #8
       BL Wyslij_dane
       ROR R4, #24
       BL Wyslij_dane

       SUBS R6, R6, #1
    ;wróć, jeśli nie ma zera
       BNE petla_Rysuj_tlo_2

    ;powrót z procedury pod adres zapamiętany na stosie
       LDR PC,[R13],#0x0004
```

Zauważmy, że zachowujemy na stosie nie tylko adres powrotu z podprogramu, ale też wartość znajdującą się w R4 (kolor). Zapamiętanie wartości rejestru R4 jest konieczne z tego względu, że używamy tego rejestru do przenoszenia parametrów do innych podprogramów (na przykład w R4 podajemy liczbę milisekund dla procedury `Czekaj_ms`). Bajty koloru są dekodowane za pomocą instrukcji `ROR`.

```
       ROR R4, #8
       BL Wyslij_dane
       ROR R4, #24
       BL Wyslij_dane
```

Pierwsze użycie instrukcji `ROR` pozwala otrzymać starszy bajt 16-bitowego koloru. Następne użycie instrukcji `ROR` z obrotem 24 bitów powoduje otrzymanie wartości rejestru R4 w pierwotnej postaci (wykonaliśmy pełny obrót bitów rejestru).

Czy zaprezentowany podprogram mógłby wyglądać inaczej? Oczywiście! Przedstawione rozwiązania nie stanowią jakiegoś ostatecznego i jedynie poprawnego algorytmu budowania kodu asemblerowego. Zapewniam Cię, szanowny Czytelniku, że wkrótce wypracujesz własny styl pisania programów w asemblerze. I bardzo dobrze, gdyż często tak jest, że tę samą rzecz można wykonać na kilka sposobów i każdy jest równie dobry.

Prawdziwym wyzwaniem jest dla nas definicja podprogramu zamalowującego określony obszar. Ponieważ współrzędne obszaru nie będą przekraczały wielkości 1 bajtu, proponuję przenosić je w rejestrze R5, według następującego schematu:

```
0x|X2|X1|Y2|Y1
```

Jak widać, w dwóch najmłodszych bajtach zostały umieszczone współrzędne Y zamalowywanego obszaru, w dwóch starszych bajtach umieściliśmy współrzędne X. Jak to działa? Załóżmy, że chcemy zapełnić kolorem obszar o współrzędnych X1 = 20, Y1 = 91, X2 = 40, Y2 = 111. Zanotujmy podane wartości w postaci szesnastkowej:

- X1 = 0x14,
- Y1 = 0x5B,
- X2 = 0x28,
- Y2 = 0x6F.

Przekazanie parametrów odbędzie się zatem za pomocą przypisania:

```
LDR R5, =0x28146F5B
```

I znów nasuwa się pytanie o to, czy nie mogliśmy parametrów obszaru przekazywać inaczej, na przykład za pomocą czterech rejestrów lub czterech bajtów pamięci RAM. Oczywiście mogliśmy. Kiedy stosuje się pewne rozwiązanie programistyczne, zawsze należy zastanowić się nad korzyściami i ewentualnymi stratami. Niewątpliwą korzyścią zaproponowanego rozwiązania jest oszczędność w zastosowaniu rejestrów. A jakie straty czy raczej — to właściwsze słowo — trudności wynikają z kodowania współrzędnych za pomocą jednego rejestru? Trudności w programowaniu, to po pierwsze. Po drugie trudności w dekodowaniu współrzędnych. Przecież będziemy chcieli na bazie podanych współrzędnych obliczyć pole obszaru do zamalowania. W tym celu konieczne będzie odczytanie wartości z R4 poprzez zastosowanie przesunięcia bitowego. Mimo wszystko uważam, że korzyści ze stosowania tylko jednego rejestru przewyższają wynikające z tego trudności.

Zanalizujmy kod obliczający pole obszaru do zamalowania. Zakładamy przy tym, że X2 ≥ X1 oraz Y2 ≥ Y1. Odpowiedni wzór znamy już z programu napisanego w języku C.

```
obszar = (X2-X1+1)*(Y2-Y1+1);
```

Zaczniemy od obliczenia członu `(X2-X1+1)`. W tym celu do rejestru R0 załadujemy wartość X2.

```
MOV R0, R5, LSR #24
```

Przed załadowaniem danych bity rejestru R4 zostały przesunięte o 24 miejsca w prawo. Skutkiem zastosowania operacji `LSR` jest wyzerowanie najstarszych bitów. Dzięki temu w rejestrze R0 znajdzie się liczba, w której ustawionych może być tylko 8 najmłodszych bitów. Do rejestru R6 załadujemy wartość X1.

```
MOV R6, R5, LSR #16
AND R6, R6, #0xFF
```

Tym razem, aby otrzymać wartość bajtową, konieczne było zastosowanie iloczynu bitowego. Obliczamy różnicę X2–X1, następnie dodajemy do niej wartość 1.

```
;R6 = R0–R6+1
SUB R6, R0, R6
ADD R6, R6, #1
```

Tę samą sekwencję działań musimy wykonać wobec współrzędnych Y1 i Y2.

```
;załaduj do R0 wartość Y2
MOV R0, R5, LSR #8
AND R0, R0, #0xFF
;załaduj do R4 wartość Y1
MOV R4, R5
AND R4, R4, #0xFF
;R0 = R0–R4+1
SUB R0, R0, R4
ADD R0, R0, #1
```

Wreszcie obliczamy pole powierzchni obszaru do zamalowania. Wynik umieścimy w rejestrze R6.

```
;R6 = R0*R6 = (Y2–Y1+1)*(X2–X1+1)
MUL R6, R0, R6
```

Cały podprogram rysowania obszaru wygląda tak:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;R4 — kolor
;R5 — współrzędne obszaru (0x|X2|X1|Y2|Y1|)
Rysuj_obszar
   ;zapamiętaj na stosie adres powrotu z procedury
      STR R14,[R13,#-0x0004]!
   ;zapamiętaj na stosie kolor
      STR R4,[R13,#-0x0004]!

   ;oblicz pole obszaru i załaduj do R6
   ;załaduj do R0 wartość X2
      MOV R0, R5, LSR #24
   ;załaduj do R6 wartość X1
      MOV R6, R5, LSR #16
      AND R6, R6, #0xFF
   ;R6 = R0–R6+1
      SUB R6, R0, R6
      ADD R6, R6, #1
   ;załaduj do R0 wartość Y2
      MOV R0, R5, LSR #8
      AND R0, R0, #0xFF
   ;załaduj do R4 wartość Y1
      MOV R4, R5
      AND R4, R4, #0xFF
   ;R0 = R0–R4+1
      SUB R0, R0, R4
      ADD R0, R0, #1
   ;R6 = R0*R6 = (Y2–Y1+1)*(X2–X1+1)
      MUL R6, R0, R6

   ;sygnalizuj rozkaz (RS = 1)
      LDR R0, =S65_RS
      STR R0, [R1]

   ;sekwencja uruchamiająca rysowanie obszaru
      LDR R4, =0xEF
      BL Wyslij_dane
      LDR R4, =0x90
      BL Wyslij_dane
      LDR R4, =0x05
      BL Wyslij_dane
      LDR R4, =0x00
      BL Wyslij_dane
      LDR R4, =0x08
      BL Wyslij_dane
      MOV R4, R5
      BL Wyslij_dane
      LDR R4, =0x09
      BL Wyslij_dane
      LSR R5, #8
      MOV R4, R5
      BL Wyslij_dane
      LDR R4, =0x0A
      BL Wyslij_dane
      LSR R5, #8
      MOV R4, R5
      BL Wyslij_dane
      LDR R4, =0x0B
      BL Wyslij_dane
      LSR R5, #8
      MOV R4, R5
      BL Wyslij_dane

   ;sygnalizuj dane (RS = 0)
      LDR R0, =S65_RS
      STR R0, [R2]
```

```
    ;odzyskaj ze stosu kolor
        LDR R4,[R13],#0x0004

    ;wypełnij kolorem
    ;w R6 jest pole obszaru
    ;LDR R6, =obszar
petla_Rysuj_obszar
    ;załaduj 2 bajty koloru
        ROR R4, #8
        BL Wyslij_dane
        ROR R4, #24
        BL Wyslij_dane

        SUBS R6, R6, #1
    ;wróć, jeśli nie ma zera
        BNE petla_Rysuj_obszar

    ;powrót z procedury pod adres zapamiętany na stosie
        LDR PC,[R13],#0x0004
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Wywołanie podprogramu powinno zawierać załadowanie do R4 koloru obszaru. W rejestrze R5 powinniśmy umieścić współrzędne obszaru. Na przykład odpowiednikiem wywołania funkcji w języku C:

```
Rysuj_obszar_S65(20, 20, 40, 40, 0, 0, 0x1F);
```

będzie taki ciąg instrukcji asemblerowych:

```
;kolor
LDR R4, =0x001F
;współrzędne obszaru
LDR R5, =0x28142814
BL Rysuj_obszar
```

Kiedy rysuje się obszary o bardziej skomplikowanym kształcie, należy odpowiednio ładować wartości współrzędnych do rejestru R4. Na przykład w taki sposób narysujemy trójkąt:

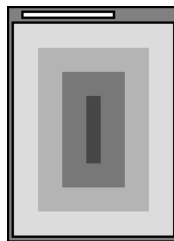
```
;narysuj czerwony trójkąt
    ;wartość początkowa R7 (k)
        LDR R7, =0
petla_trojkat
;for(k=0; k<40; k++)
;     Rysuj_obszar_S65(50+k, 66-k, 50+k, 66+k, 0x1F, 0, 0);
     ;kolor
          LDR R4, =0xF800
        ;współrzędne obszaru
     ;X2
          LDR R5, =50
     ;dodaj wartość k
          ADD R0, R5, R7
     ;kopiuj wynik do X2
          MOV R5, R0
     ;przesuń o 8 miejsc w lewo
          LSL R5, #8
        ;X1
          ORR R5, R5, R0
     ;przesuń o 8 miejsc w lewo
          LSL R5, #8
     ;Y2 = 66+k
          ORR R5, R5, #66
          ADD R5, R5, R7
     ;przesuń o 8 miejsc w lewo
          LSL R5, #8
     ;Y1 = 66–k
          ORR R5, R5, #66
          SUB R5, R5, R7
          BL Rysuj_obszar

     ;inkrementuj zmienną pętli
          ADD R7, R7, #1
```

```
;sprawdź, czy koniec pętli
     CMP R7, #40
;wróć, jeśli mniejsze
     BCC petla_trojkat
```

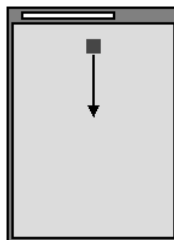
15.3. Ćwiczenia

1. Narysuj na wyświetlaczu S65 cztery zagnieżdżające się prostokąty (patrz rysunek 15.15).



Rysunek 15.15. *Rysunek mający się pojawić na wyświetlaczu S65 w wyniku wykonania ćwiczenia 1.*

2. Zaprogramuj wyświetlacz S65 tak, aby na matrycy widać było poruszający się w pionie kwadrat (patrz rysunek 15.16).



Rysunek 15.16. *W wyniku wykonania ćwiczenia 2. na wyświetlaczu powinien pojawić się kwadrat poruszający się w pionie (z góry na dół i z dołu do góry)*

AVR & ARM7

PROGRAMOWANIE MIKROKONTROLERÓW DLA KAŻDEGO

Jeśli nie masz pojęcia o programowaniu mikrokontrolerów, a chcesz się tego nauczyć, ta książka jest właśnie dla Ciebie. Nie musisz wcześniej mieć wiedzy z zakresu elektroniki, ponieważ wszystkie potrzebne pojęcia zostały tu wyjaśnione od podstaw. Niepotrzebna Ci także znajomość programowania w jakimkolwiek języku — te informacje, podane w możliwie najbardziej przystępny sposób, też znajdziesz w podręczniku. Wobec tego wszystko, czego potrzebujesz, to chęć nauki. I jeszcze jedno: może zastanawiasz się, co począć z takim mikrokontrolerem? Otóż możesz zastosować go do konstruowania efektów świetlnych z diod, sterowania modelami samolotów, a nawet sterowania robotami.

Jeśli wiesz już co nieco na temat programowania mikrokontrolerów, ale chcesz poszerzyć swoją wiedzę — do tego również przyda się ta książka. Dzięki niej dowiesz się, na czym polega programowanie mikrokontrolerów z dwóch rodzin: AVR (na przykładzie układu ATmega8) i ARM7 (na przykładzie układu LPC2106). Nauczysz się programowania układów w czterech językach programowania: asemblerze (środowisko AVR Studio 4), języku C (środowisko WinAVR), języku bascom (środowisko Bascom) oraz Pascalu (środowisko mikroPascal). Z łatwością zdobędziesz, a potem — wykonując poszczególne ćwiczenia — sprawdzisz nowe, niesamowite umiejętności, ponieważ cała wiedza podana jest tu przejrzyście i w dodatku z humorem.

- Programowanie mikrokontrolerów z rodziny AVR oraz ARM7
- Obsługa wyświetlaczy graficznych z telefonu komórkowego Siemens S65
- Obsługa diod i wyświetlaczy LED
- Komunikacja między mikrokontrolerami (USART)
- Obsługa przycisków i klawiatur
- Serwomechanizmy
- Wyświetlacze alfanumeryczne
- Kompilatory
- Obsługa przerwań
- Programowanie z użyciem systemów czasu rzeczywistego na przykładzie

Cała wiedza potrzebna, aby zostać ekspertem od programowania mikrokontrolerów!

Cena: 77,00 zł

helion.pl
księgarnia internetowa

Nr katalogowy: 5424

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne:

0 801 339900

0 601 339900

Zamów najnowszy katalog: http://helion.pl/katalog
Zamów informacje o nowościach: http://helion.pl/nowosci
Zamów cennik: http://helion.pl/cennik

Wydawnictwo Helion
ul. Kościuszki 1c, 44-100 Gliwice
44-100 Gliwice, skr. poczt. 462
032 230 98 63
http://helion.pl
e-mail: helion@helion.pl

ISBN 978-83-246-2628-1

Informatyka w najlepszym wydaniu