Tetris na STM32Butterfly (2)

W cyklu artykułów przedstawię proces tworzenia klonu gry Tetris na platformę z mikrokontrolerem 32-bitowym z rodziny STM32 firmy STMicroelectronics. W części drugiej opisuję mechanizm gry oraz jego realizację w języku C.

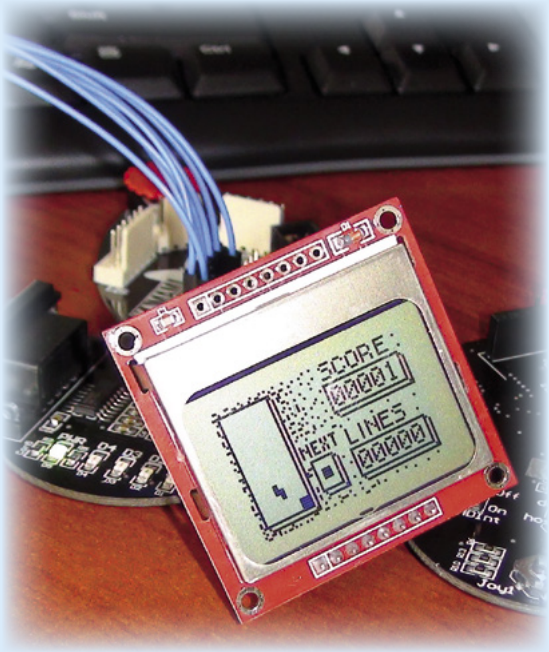
Przygotowana przeze mnie gra polega na takim układaniu spadających klocków, aby ich segmenty tworzyły ciągłe poziome linie na planszy. Każda ułożona linia znika robiąc miejsce dla kolejnych klocków. W mojej wersji gry plansza ma wymiary 10 na 20 pól. Planszę w pamięci mikrokontrolera będzie odzwierciedlała tablica zmiennych unsigned char plansza[10][20];

Każda komórka tej tablicy reprezentuje jedno pole planszy. Jeżeli komórka przechowuje 0, to znaczy, że pole jest puste. Wartość komórki równa 1 sygnalizuje, że w danym polu znajduje się segment klocka. Współrzędna 0,0 będzie wskazywała na pole w lewym, górnym narożniku planszy. Ktoś zaraz wzburzy się na widok tak beztrosko marnowanej pamięci. Jest to jednak działanie celowe. Mamy z tego dwie korzyści: znaczne uproszczenie programu, bowiem artykuł skierowany jest głównie do początkujących oraz możliwość przyszłej rozbudowy programu, np. klocki na kolorowym wyświetlaczu mogą mieć różne kolory.

Myślę, że pierwszy fragment programu, który napiszemy powinien zerować naszą tablicę. Umieśćmy go w osobnej funkcji, tak aby było nam wygodnie wykonać tę czynność z dowolnego fragmentu naszej gry (**listing 3**). W funkcji użyłem dwóch pętli for do odliczania współrzędnych x i y. Do każdej komórki jest zapisywane 0.

W tablicy będzie odwzorowana tylko statyczna część planszy, czyli klocki, które już opadły i zatrzymały się na dnie planszy. Potrzebujemy jeszcze zmiennych, które będą przechowywały informacje o klocku, który spada. Do tego celu przeznaczone będą zmienne unsigned char klocekx, kloceky, klocekr; przechowujące współrzędne x i y klocka na planszy oraz informacje o kącie jego obrotu. Zmienna klocekr może przyjmować jedną z czterech wartości, od 0 do 3, które odpowiadają obrotowi o 0, 90, 180 i 270 stopni. Wartość kolejnej zmiennej unsigned char kloceknr; odpowiada jednemu z 7 różnych kształtów spadających klocków. Jak zatem program odwzorowuje kształt klocka na planszy? W pamięci programu znajduje się tablica stałych przechowująca współrzędne każdego segmentu każdego rodzaju klocka w każdym z 4 pozycji obrotu. Tablica ma postać const unsigned char klocki[7][4][4][2];. Pierwszy indeks tablicy przechowuje rodzaje klocków, kolejny położenie klocka (kąt obrotu), następny segment klocka i jeszcze jeden – jego współrzędne x i y. Fragment tablicy opisujący kształt jednego z klocków pokazano na **rysunku 8**.

Wbrew pozorom, korzystanie z takiej tablicy w programie jest bardzo łatwe. Dodanie współrzędnej x danego segmentu odczytanej z tablicy do współrzędnej x całego klocka da nam położenie x tego segmentu na planszy. Dodatkowym ułatwieniem jest fakt, że liczba segmentów dla każdego z 7



Listing 3. Zerowanie tablicy – czyszczenie planszy

```
void CzyscPlansze(void)
{
  unsigned char x,y;

  for(y=0;y<20;y++)  //odliczanie wierszy
    for(x=0;x<10;x++)  //odliczanie kolumn
      plansza[x][y] = 0;  //wypełnianie komórek zerami
}
```

rodzajów klocków jest stała i wynosi 4. Teraz wystarczy zmieniać współrzędne klocka, aby zmieniać położenie na planszy każdego z jego 4 segmentów. Np. aby klocek spadał w dół należy cyklicznie zwiększać o 1 jego współrzędną y.

Jednak skąd będziemy widzieli kiedy klocek oprze się na istniejących już segmentach na planszy i nie powinien spadać dalej? Należy zbudować funkcję, która będzie sprawdzała czy wystąpiła kolizja klocka z segmentami na planszy. Trzeba sprawdzić kolejno każdy z segmentów klocka, do tego posłuży nam pętla:

```
for(i=0;i<4;i++)
{
  ...
}
```

W każdej iteracji pętli należy obliczyć współrzędne x i y danego segmentu klocka na planszy:

```
x=klocki[kloceknr][r][i][0]+kx-2;
y=klocki[kloceknr][r][i][1]+ky-2;
```

Gdzie *kx*, *ky*, *r* to położenie klocka i sprawdzić czy nie ma w tym miejscu położonego już na stałe segmentu:

```
if(plansza[x][y]) kolizja=1;
```

W wypadku, gdy równanie w warunku jest prawdziwe (wystąpiła kolizja) ustawiana jest zmienna kolizja zwracana później w wyniku działania funkcji. W tej samej funkcji umieścimy także warunki sprawdzające czy klocek nie wystaje poza boczne ścianki planszy:

```
if(x > 9) kolizja=1;
```

…i czy nie przeszedł poniżej dna:

```
if(y > 19) kolizja=1;
```

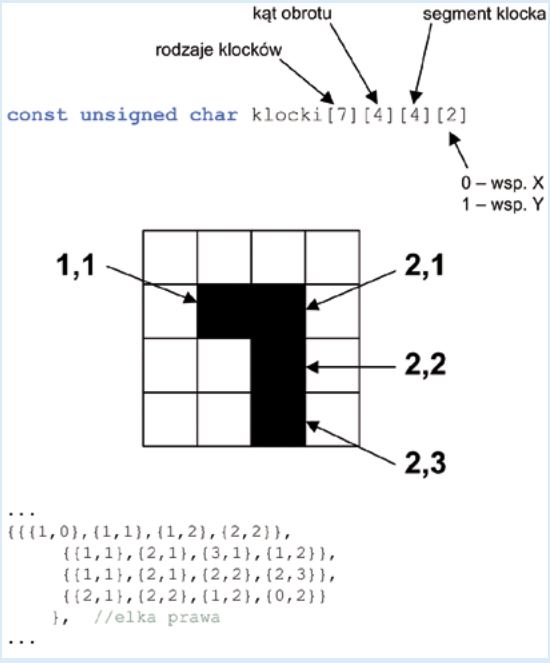
Gotowa funkcja będzie wyglądała jak na **listingu 4**. Teraz wystarczy, że przed każdym kolejnym ruchem klocka będziemy sprawdzać czy nie wywoła od kolizji. Np. przed opuszczeniem klocka w dół sprawdzimy następujący warunek `if(Kolizja(klocekx,kloceky+1,klocekr) == 0) kloceky++;`

Konstrukcja taka zwiększy zmienną kloceky o jeden, tylko w przypadku, kiedy funkcja sprawdzająca zwróci wartość 0. Podstawiając do funkcji współrzędna y klocka zwiększona o jeden sprawdzamy czy opuszczenie klocka spowoduje kolizję, zanim opuścimy klocek. Analogicznie będziemy sprawdzać wystąpienie kolizji podczas wykonywania klockiem ruchów na boki.

Przygotujmy zatem komplet funkcji odpowiedzialnych za poszczególne ruchy klocka. Pierwsza funkcja przemieszcza klocek o jedno pole w lewo (**listing 5**). Proste, prawda? Myślę, że już każdy wie, jak będzie wyglądała funkcja przemieszczająca klocek w prawo. Równie łatwe będzie obracanie klocka. Wystarczy tylko zmienić wartość jednej zmiennej, na kolejną z 4 wartości, które może przyjąć (**listing 6**).

Mam nadzieję, że już widzicie zalety proponowanego rozwiązania. Pozostała nam jeszcze do napisania funkcja przemieszczająca klocek o jedno pole w dół. Tutaj sytuacja będzie nieco bardziej skomplikowana. Zastanówmy się jak nasz program powinien zareagować, jeżeli podczas opuszczania klocka kolejny ruch nie będzie możliwy. Klocek, który do tej pory był w ruchu powinien zostać odwzorowany na planszy w postaci stałych jej segmentów. Napiszmy wiec funkcję, która będzie tego dokonywała. Zamieszczono ją na **listingu 7**. Podobnie jak w przypadku sprawdzania kolizji zbudujemy pętlę odliczającą kolejne segmenty klocka, wewnątrz której będziemy obliczać ich współrzędne na planszy. Z tą jednak różnicą, że ustawimy odpowiednie komórki w tablicy odwzorowującej planszę.

Kolejnym działaniem po odwzorowaniu klocka na planszę będzie wygenerowanie kolejnego klocka. Będzie to polegało na ustawieniu współrzędnych klocka na położenie początkowe. Wyzerowaniu zmiennej obrotu klocka, oraz zmiany rodzaju klocka na inny, najlepiej w sposób losowy. Tworzymy kolejna funkcję, która będzie za to odpowiedzialna. Pokazano ją na **listingu 8**.



Rysunek 8. Fragment tablicy opisujący kształt znaku „prawe L"

Listing 4. Funkcja testująca klocek

```
unsigned char Kolizja(unsigned char kx, unsigned char ky, unsigned
char r)
{
  unsigned char x,y,i,kolizja;
  kolizja=0;

  for(i=0;i<4;i++)  //pętla odlicza kolejne cztery segmenty klocka
  {  //(każdy klocek składa się z czterech segmentów)
    x=klocki[kloceknr][r][i][0]+kx-2;  //obliczanie współrzędnych
    y=klocki[kloceknr][r][i][1]+ky-2;  //danego segmentu klocka
//sprawdzenie czy segment jest poza ściankami bocznymi planszy
    if(x > 9) kolizja=1;
//sprawdzenie czy segment jest poniżej dna planszy
    if(y > 19) kolizja=1;
//sprawdzenie czy segment nie nakłada się na punkt istniejący na
planszy
    if(plansza[x][y]) kolizja=1;
  }
//jeżeli wykryto kolizję to funkcja zwraca 1 w innym razie 0
  return kolizja;
}
```

Listing 5. Przemieszczenie klocka o jedno pole w lewo

```
void KlocekLewo(void)
{
//sprawdzenie czy przesunięty klocek nie wywoła kolizji
  if(Kolizja(klocekx-1,kloceky,klocekr) == 0)
//jeżeli nie to zmniejsz zmienną x położenia klocka
  {
    klocekx--;
  }
}
```

Listing 6. Funkcja obracająca klocek

```
void KlocekObroc(void)
{
//sprawdzenie czy obrócony klocek nie wywoła kolizji
  if(Kolizja(klocekx,kloceky,klocekr+1) == 0)
  {
// jeżeli nie to zwiększ zmienną obrotu klocka
    klocekr++;
    if(klocekr > 3) klocekr = 0;
  }
}
```

Listing 7. Funkcja przemieszczająca klocek

```
void KlocekWklej(void)
{
  unsigned char x,y,i;
//pętla odlicza kolejne 4 segmenty klocka
  for(i=0;i<4;i++)
  {
//oblicz położenie segmentu na planszy
    x=klocki[kloceknr][klocekr][i][0] + klocekx-2;
    y=klocki[kloceknr][klocekr][i][1] + kloceky-2;
//zapal punk na planszy w miejscu segmentu klocka
    plansza[x][y]=1;
  }
}
```

Listing 8. Funkcja generująca nowy klocek

```
void KlocekNowy(void)
{
//kopiuj rodzaj klocka jaki był następny jako bieżący
  kloceknr=nkloceknr;
//kopiuj rodzaj następnego klocka z zmiennej losującej
  nkloceknr=rkloceknr;
//jeżeli następny klocek jest taki sam jak bieżący
  if(kloceknr==nkloceknr)
  {
    rkloceknr++;  //zwiększ zmienną losującą
    if(rkloceknr>6) rkloceknr=0;
    nkloceknr++;  //zwiększ zmienną z rodzajem
kolejnego klocka
    if(nkloceknr>6) nkloceknr=0;
  }
  klocekr=0;  //zerowanie położenia nowego klocka na
planszy
  klocekx=5;
  kloceky=2;
}
```

Pojawiają się tutaj nowe zmienne: nkloceknr i rkloceknr, należy je zadeklarować globalnie. Pierwsza z nich określa, jaki klocek będzie następny po bieżącym. Pozwoli nam to na wyświetlenie na ekranie nadchodzącego klocka. Następna to zmienna z losowym rodzajem klocka. Jak sprawić, żeby w zmiennej znajdowała się losowa liczba? Musimy skorzystać z pewnego uproszczenia. Prawie każdy kompilator języka ma funkcję generatora liczb losowych. Liczby te są generowane pseudolosowo, według ustalonych schematów, np. za pomocą zmiennej, która jest zwiększana cyklicznie w jakimś przerwaniu. Losowość takiej liczby zwiększa się głównie dzięki interakcji urządzenia z użytkownikiem. Zdecydowałem się na napisanie własnej funkcji generatora liczba pseudolosowych i wykorzystałem rozwiązanie z inkrementacją zmiennej w przerwaniu timera SysTick, które daje w wypadku nieskomplikowanej gry bardzo dobre rezultaty. Ponieważ uruchomiliśmy już ten timer wcześniej, wystarczy że dodamy procedurę cyklicznej zmiany zmiennej rkloceknr do funkcji SysTick_Handler:

```
rkloceknr++;  //losowanie klocków
if(rkloceknr>6) rkloceknr=0;
```

Ponieważ funkcje obsługi przerwań znajdują się w pliku stm32f10x_it.c, a zmienna rkloceknr została zadeklarowana w pliku main.c, musimy poinformować o tym fakcie kompilator dopisując na początku funkcji SysTick_Handler:

```
extern unsigned char rkloceknr;
```

Kolejny klocek pojawia się na naszej planszy i jest gotowy do ułożenia przez gracza na dnie. Jednak co, jeśli układana piramida sięgnie samej góry planszy i zbraknie miejsca na pojawienie się nowego klocka? Gra kończy się powodując większą lub mniejszą frustrację gracza. Dodajmy zatem do funkcji KlocekNowy fragment kodu, który sprawdzi czy pojawienie się nowego klocka nie wywoła od razu kolizji. Korzystamy w tym celu z wcześniej napisanej funkcji Kolizja (**listing 9**).

Pojawiły się tu nowe zmienne, których użycie wymaga wyjaśnień. Najoczywistsze będzie przeznaczenie zmiennej punkty. Przechowuje ona ilość punktów zdobytych przez gracza. Jeżeli nowy klocek mieści się na planszy punktacja zwiększana jest o jeden.

Natomiast zmienna stan_gry określa, w jakim, z możliwych czterech, stanie jest gra. Na początku pliku z programem dodajemy deklaracje nazw tych stanów:

```
#define INTRO 1
#define GRA 2
#define GAMEOVER 3
#define PAUZA 4
```

Po uruchomieniu gra wyświetla planszę początkową, a zmienna stan_gry ma wartość INTRO. Dopiero naciśnięcie przycisku przez użytkownika rozpoczyna właściwą grę, zmienna zmienia stan na wartość GRA, w trakcie pauzy jej wartość przyjmie wartość PAUZA, a po zakończeniu gry GAMEOVER. Oczywiście, nie obędzie się bez zadeklarowania naszych nowych zmiennych jako globalnych na początku pliku *main.c*:

```
//Zmienna określa rodzaj spadającego klocka
unsigned char kloceknr;
//Zmienna określa rodzaj następnego klocka
unsigned char nkloceknr;
//Zmienna na potrzeby losowania rodzaju
klocka
unsigned char rkloceknr;
//Zmienna określająca stan programu
unsigned char stan_gry=INTRO;
//Punkty
unsigned int punkty;
```

Ułożyliśmy, zatem klocek na planszy oraz wygenerowaliśmy kolejny, gotowy do ułożenia. Jednak, aby gra miała sens musimy premiować w postaci punktów ułożenie kompletnej linii poziomej. Poza tym kompletne linie należy usuwać, aby zrobić miejsce na kolejne, spadające klocki. Potrzebujemy zatem kolejnej funkcji, która się tym zajmie.

Cała plansza to 20 linii poziomych, które należy sprawdzić po kolei. Zaczynamy więc od pętli odliczającej kolejne linie:

```
for(y=0;y<20;y++)
```

W każdej iteracji pętli musimy sprawdzić czy dana linia zawiera wszystkie segmenty, których jest 10. Czyli kolejna pętla. Ja proponuję następującą konstrukcję:

```
k=1;
for(x=0;x<10;x++) if(plansza[x][y] == 0)
k=0;
```

Do zmiennej p wpisujemy na początku wartość 1. Teraz w każdym przejściu pętli sprawdzany jest jeden z dziesięciu segmentów linii. Brak jednego z segmentów spowoduje zapisanie do zmiennej p wartości 0. Więc wartość 1 zmiennej p świadczy o tym, że bieżąca linia jest kompletna.

Co jeżeli trafimy na pełną linię? Musimy usunąć ją z planszy. Dokonamy tego kopiując każdą kolejną linię nad tą, którą chcemy skasować, do linii poniżej. Odliczanie linii rozpoczniemy od aktualnie sprawdzanej:

```
for(yy=y;yy>0;yy--)
```

Listing 9. Testowanie warunku zakończenia gry

```
//sprawdzanie czy nowy klocek nie spowoduje kolizji
  if(Kolizja(klocekx,kloceky,klocekr) == 0)
  {  //jeżeli nie zwiększ punktację
    punkty++;
  }else  //jeżeli nowy klocek wywołał od razy kolizję
  {
    stan_gry=GAMEOVER;  //koniec gry!
  }
```

Listing 10. Funkcja usuwająca wypełnione linie

```
void Redukcja(void)
{
  unsigned char x,y,k,yy;
  for(y=0;y<20;y++) //pętla oblicza kolejne linie planszy
  {
    k=1;
//pętla sprawdza każdy punkt danej linii
    for(x=0;x<10;x++) if(plansza[x][y] == 0) k=0;
//jeżeli brakuje któregoś klocka to k=0;
    if(k == 1)  //jeżeli bieżąca linia jest kompletna
    {
      linie++;  //zwiększ punkty
      punkty+=10;
//pętla odlicza kolejne linie od bieżącej w górę
      for(yy=y;yy>0;yy--)
      {
//pętla kopiuje linię yy-1 do linii yy
        for(x=0;x<10;x++)
//spowoduje to skasowanie pełnej linii i przesunięcie wszystkich
linii ponad w dół
          plansza[x][yy]=plansza[x][yy-1];
      }
    }
  }
}
```

Listing 11. Funkcja opuszczająca klocek o jedno pole w dół

```
void KlocekDol(void)
{
//sprawdzenie czy przesunięty klocek nie wywoła kolizji
  if(Kolizja(klocekx,kloceky+1,klocekr) == 0)
  {
//jeżeli nie to zwiększ zmienną y położenia klocka
    kloceky++;
  }else
//jeżeli wykryto kolizję skopiuj segmenty klocka na planszę
  {
    KlocekWklej();
    Redukcja();  //sprawdź czy nie ma pełnych linii
    KlocekNowy();  //załaduj nowy klocek
  }
}
```

Listing 12. Deklaracja tablicy tlo[]
```
const unsigned char tlo[]={
  0xC4,0x15,0xFD,0x06,0x04,0x04,0x05,0x05,0x04,0x06,
  0x06,0x04,0x05,0x06,0x04,0x04,0x06,0x05,0x04,0x06,
  0x06,0x05,0x04,0xFD,0x0C,0xF2,0x04,0x35,0x58,0x44,
  0x84,0x11,
.
.  //kolejne 504 bajty grafiki (fragment)
.
  0x0A,0x0A,0x0A,0x2B,0x0C,0x0F,0x00,0x00
  };
```

Listing 13. Deklaracje stałych używanych w programie
```
#define POFFX 3      //wsp. X górnego lewego rogu planszy
#define POFFY 3      //wsp. Y górnego lewego rogu planszy
#define NKOFFX 30    //wsp. X górnego lewego rogu miejsca
wyświetlania następnego klocka
#define NKOFFY 31    //wsp. Y górnego lewego rogu miejsca
wyświetlania następnego klocka
#define WOFFX 9      //wsp. X wyniku
#define WOFFY 2      //wsp. Y wyniku
#define LOFFX 9      //wsp. X ilości linii
#define LOFFY 5      //wsp. Y ilości linii
```

Listing 14. Rysowanie stałych segmentów klocków
```
for(y=0;y<20;y++)  //odliczanie wierszy pamięci planszy
  for(x=0;x<10;x++)  //odliczanie kolumn
  {
//jeżeli dany punkt na planszy jest zapalony
   if(plansza[x][y])
   {
//rysowanie kolejnych 4 pikseli punktu
    LcdPixel(OFFX+x*2, POFFY+y*2, PIXEL_ON);
    LcdPixel(OFFX+x*2+1, POFFY+y*2, PIXEL_ON);
    LcdPixel(OFFX+x*2, POFFY+y*2+1, PIXEL_ON);
    LcdPixel(OFFX+x*2+1, POFFY+y*2+1, PIXEL_ON);
   }
  }
```

Teraz wystarczy przenieść kolejne wartości każdej z 10 komórek z linii y-1, czyli tej wyżej:

```
for(x=0;x<10;x++) plansza[x][yy]=plansza[x]
[yy-1];
```

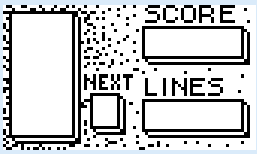
Za usuniętą linię należą się graczowi punkty. W naszej wersji gry zrealizowałem zliczanie punktów: 1 punkt za każdy ułożony klocek, 10 punktów za każdą kompletną linię która zniknie. Punkty przechowuje zmienna punkty, ponadto w zmiennej linie zliczane są ułożone linie. Kompletna funkcja będzie zatem wyglądała jak pokazano na **listingu 10**. Kolejna nowa zmienna linie zlicza ilość ułożonych przez gracza linii poziomych, należy ją zadeklarować tak samo jak zmienną punkty.

Napisaliśmy wszystkie funkcje potrzebne podczas opuszczania klocka, więc czas na samą funkcję KlocekDol – zamieszczono ją na **listingu 11**.

Dzięki temu, że rozbiliśmy poszczególne czynności na odrębne kawałki kodu, cały program staje się bardzo czytelny i łatwo nam będzie w przyszłości dokonywać w nim zmian. Zapewne każdy z czytelników będzie chciał zmodyfikować grę po swojemu, dlatego starałem się pisać ją jak najprzejrzyściej, a zarazem „rozwojowo".

Mamy więc komplet funkcji odpowiedzialnych za mechanikę gry, natomiast brakuje istotnego elementu jakim jest interfejs użytkownika. Interfejs ten będzie składał się z dwóch części, obsługi joysticka i wyświetlacza graficznego. Dzięki przygotowanej wcześniej obsłudze przerwań

Listing 15. Rysowanie spadającego klocka
```
  //rysownie spadającego klocka
  for(i=0;i<4;i++)  //odliczanie 4 segmentów klocka
  {
//obliczanie współrzędnych na planszy
    x=klocki[kloceknr][klocekr][i][0] + klocekx-2;
//danego segmentu klocka
    y=klocki[kloceknr][klocekr][i][1] + kloceky-2;
//rysowanie kolejnych 4 pikseli segmentu
    LcdPixel(POFFX+x*2, POFFY+y*2, PIXEL_ON);
    LcdPixel(POFFX+x*2+1, POFFY+y*2, PIXEL_ON);
    LcdPixel(POFFX+x*2, POFFY+y*2+1, PIXEL_ON);
    LcdPixel(POFFX+x*2+1, POFFY+y*2+1, PIXEL_ON);
   }
```



z pinów podłączonych do styków joysticka, bardzo łatwo będzie oprogramować tę część interfejsu.

Na początku zajmijmy się poruszaniem klocka na boki. Będzie to bardzo łatwe, bo mamy już gotowe funkcje, które to robią. Przed ich wywołaniem musimy tylko sprawdzić czy w zmiennej stan_gry zapisana jest wartość GRA. Zapobiegnie to przesuwaniu klocka podczas pauzy lub po zakończeniu gry. W funkcji przerwania EXTI15_10_IRQHandle w obrębie warunku sprawdzającego czy joystick był naciśnięty w prawo, dodajemy:

```
if(stan_gry==GRA)  //Jeżeli gra włączona
{
  KlocekPrawo();  //przesuń klocek w prawo
}
```

Analogiczny wpis dodajemy dla ruchu w lewo i dół, nie zapominając o dodaniu deklaracji (z modyfikatorem extern) zmiennej stan_gry na początku funkcji.

Do funkcji EXTI9_5_IRQHandler dodamy natomiast wywołanie funkcji obracającej klocek, jeżeli joystick zostanie przesunięty w górę. Tu także dodajemy definicję zmiennej stan_gry. Natomiast na początek pliku z funkcjami przerwań kopiujemy deklaracje nazw wartości dla tej zmiennej (NITRO, GRA… idt.).

Obsługę joysticka mam już wstępnie opanowaną, musimy dodać jeszcze to i owo, ale to w późniejszym czasie. Zajmijmy się teraz tym, co powinno się dziać na wyświetlaczu. Do jego obsługi można zaadoptować jedna z wielu dostępnych gotowych bibliotek. Ja napisałem swoją od podstaw tak, aby była jak najprostsza. Nie będę tu wnikał zbytnio w szczegóły, zainteresowani sami prześledzą funkcje wyświetlania grafiki. Jeżeli chcesz skorzystać z mojej biblioteki to dodaj pliki NokiaLCD.c i NokiaLCD.h do projektu, a na początku pliku *main.c* trzeba dodać #include "NokiaLCD.h". W ten sposób kompilator uzyska informację o wszystkich funkcjach biblioteki. Przed korzystaniem z wyświetlacza należy jeszcze wywołać funkcje LcdInit. Przygotowuje ona sprzęt do pracy. Najlepiej zrobić to zaraz po konfiguracji peryferiów mikrokontrolera.

Skupmy się teraz na funkcji, która rysuje ekran z planszą gry. Ja w swoim projekcie nazwałem ją RysujEkran. Na początek wywołujemy funkcję LcdClear z naszej biblioteki graficznej, w celu usunięcia poprzedniej zawartość ekranu. W kolejnym kroku rysujemy na ekranie wszystkie stałe elementy tła. Ja takie tło zaprojektowałem w Paint, następnie zapisałem w 2 kolorowym formacie BMP i użyłem jednego w wielu dostępnych w sieci programów do konwersji bitmap na tablice zgodne z C. W katalogu projektu stworzyłem nowy plik *grafika.h* i tak skopiowałem tablicę nazywając ją tlo[]. Jej deklarację pokazano na **listingu 12**. Tak przygotowaną grafikę wyświetla funkcja LcdLoadBMP(tlo);

Teraz wystarczy narysować samą planszę, spadający klocek, klocek jaki będzie następny oraz wynik gracza. Jednak zanim napiszemy resztę funkcji proponuję zdefiniować współrzędne każdego z tych elementów. Przyda się to nam jak będziemy chcieli w przyszłości zmienić układ graficzny. Swoje deklaracje zamieściłem na **listingu 13**.

Najpierw narysujemy stałe segmenty klocków na planszy. Przy tak niewielkiej rozdzielczości wyświetlacza postanowiłem, że każdy segment klocka będzie składał się z czterech pikseli. Aby narysować całą planszę musimy sprawdzić każdą z komórek tablicy plansza, czyli zastosujemy dwie

pętle i warunek. Jeżeli dana komórka będzie miała wartość 1 to należy zapalić odpowiadające jej cztery piksele na ekranie. Ten fragment kodu napisałem w sposób pokazany na **listingu 14**. Do rysowania pojedynczych pikseli służy funkcja LcdPixel. Przyjmuje ona trzy argumenty: współrzędną X, współrzędną Y oraz wartość *PIXEL_ON* lub *PIXEL_OFF* w zależności czy zapalamy czy gasimy dany piksel. Przy obliczaniu współrzędnych korzystamy z zmiennych x i y z pętli. Mnożymy te wartości przez 2, ponieważ każdy segment planszy ma wymiar 2x2, oraz dodajemy wcześniej zdefiniowany offset oraz ewentualne przesunięcie o 1 w obrębie danego segmentu.

Analogicznie narysujemy spadający klocek, jednak tym razem użyjemy jednej pętli odliczającej 4 kolejne jego segmenty (**listing 15**). Tym razem, aby obliczyć współrzędne danego segmentu musimy sięgnąć do tablicy definiującej klocek podając numer kształtu klocka oraz jak jest obrócony. Następnie dodać offset planszy oraz współrzędną klocka na planszy. Ponieważ współrzędne klocka umownie wskazują jego środek odejmujemy jeszcze od wszystkiego wartość 2. Tak obliczoną wartość x i y wykorzystujemy do narysowania czterech pikseli danego segmentu klocka.

Identycznie narysujemy drugi klocek w okienku z podpisem NEXT. Jest to podpowiedź dla gracza, jakiego następnego klocka ma się spodziewać. Zmiany będą polegały na wykorzystaniu innych offsetów oraz zmiennej nklockeknr zamiast klockeknr.

Zostało nam jeszcze wyświetlenie wyniku gry. Typowe biblioteki graficzne oferują nam zazwyczaj funkcję wyświetlającą pojedyncze znaki ASCII. Ja mam w swojej bibliotece właśnie taką funkcję: LcdChr(*znak_ascii*) oraz funkcję LcdGotoXY(*wsp.x*, *wsp.y*) do ustawiania kursora na daną pozycję. Przy czym rysowanie kolejnych znaków automatycznie przesuwa kursor do przodu. Jak widać konwersję liczby, na znaki ASCII musimy zrobić sobie sami. Zastosowałem tu bardzo prosty algorytm zbudowany na jednej pętli for (**listing 16**).

W kolejnych iteracjach wartość skopiowana do zmiennej val jest dzielona kolejno przez 10000, 1000, 100 i 10. po każdej takiej operacji wynik zapamiętany jest z zminnej c, a obliczone dziesiątki tysięcy, tysiące itd., odejmowane są od zmiennej aby w kolejnym wykonaniu pętli nie wpływały na obliczenie wartości kolejnej pozycji dziesiętnej. W celu konwersji do zmiennej c dodawana jest wartość ASCII znaku „0" (zero). Kolejne miejsca dziesiętne wyświetlane są na ekranie. W tym prostym algorytmie celowo nie wygaszam zer wiodących (tych na początku, których normalnie nie piszemy), aby gra miała swoisty klimat 8-bitowców.

Identyczny fragment programu wyświetla ilość zaliczonych linii. Oczywiście na innym offsecie. Ponieważ wszystkie dotychczasowe operacje graficzne były dokonywane na buforze, funkcja wyświetlająca kończy się wywołaniem funkcji LcdUpdate, która przenosi zawartość bufora na ekran.

W tym miejscu na pewno każdy nie myśli o niczym innym, jak tylko o tym, aby w końcu wypróbować to, co do tej pory napisaliśmy. Skompilowanie i uruchomienie programu z niemal pustą funkcją main na pewno jest bezcelowe. Musimy dodać główną pętlę programu, w której zawartość ekranu będzie cyklicznie odświeżana w zależności od wartości zmiennej stan_gry. Proponuję dodać do funkcji main procedurę pokazaną na **listingu 17**. W nieskończonej pętli sprawdzana jest wartość nowej zmiennej refresh. Pozwoli to dać sygnał z innych części programu, kiedy należy odświeżyć zawartość ekranu i zapobiegnie wykonywaniu tej czynności w kółko nawet wtedy, gdy nie jest to konieczne. Aby wygodnie było operować zmienną refresh dodałem funkcję, która ją ustawia:

```
//Funkcja ustawia zmienną refresh
void OdswiezEkran(void)
{
  refresh=1;
}
```

Wracając do naszej funkcji main. W konstrukcji opartej na switch program sprawdza wartość zmiennej stan_gry, ponieważ nie zawsze zawartość wyświetlacza będzie taka sama. Na razie program zadziała tylko w czasie rozgrywki, wyświetlając planszę gry. Z czasem dodamy funkcję wyświetlającą intro oraz komunikaty o pauzie i zakończeniu gry. Podobną konstrukcję opartą o polecenie switch dodamy do obsługi klawisza „ok" (naciśnięcie z góry) joysticka (**listing 18**). Jeżeli naciśniemy „ok." w trakcie trwania intro, to uruchomi się gra, w czasie gry uruchomimy w ten sposób pauze, a po zakończeniu gry powrócimy do intro uprzednio kasując punkty i czyszcząc planszę. Na końcu uruchamiana jest funkcja OdswiezEkran, aby wszystkie zmiany widoczne były natychmiast na ekranie. Jej wywołanie musimy dodać na końcu każdego fragmentu obsługującego poszczególne ruchy joysticka.

Wypróbujmy w końcu nasz program. Przed główną pętlą programu dodajmy jeszcze:

```
  CzyscPlansze();  //Czyszczenie planszy
  KlocekNowy();  //Załadowanie nowego klocka
//Zerowanie liczników z punktami
  punkty=0;
  linie=0;
  stan_gry=GRA;
  refresh=1;
```

Spowoduje to, że od razu znajdziemy się w trybie gry. Skompiluj program i spróbuj go uruchomić. Działa? No, prawie… Możemy poruszać klockiem, punkty są naliczane, linie znikają… Wszystko byłoby dobrze, gdyby klocek chciał sam spadać. Nic prostszego, wystarczy cyklicznie wywoływać funkcję KlocekDol w przerwaniu timera SysTick. Tak też zrobimy, jednak, aby gra nabrała smaczku i nie była nudna

Listing 16. Funkcja pozycjonująca kursor na ekranie

```
LcdGotoXY(WOFFX, WOFFY);
val=punkty;
//odliczanie kolejnych miejsc dziesietnych
for(d=10000;d>0;d=d/10)
{
  c=val/d;  //dzielenie wyniku na tysiące, setki itd.
  val=val-(c*d);  /*odejmowanie tysięcy, setek, itd. od wyniku,
aby można było obliczyć kolejne miejsce dziesiętne w kolejnym
kroku pętli */
  c=c+'0';  //konwersja wyniku na cyfrę w kodzie ASCII
  LcdChr(c);  //wyświetlanie kolejnych cyfr wyniku na ekranie
}
```

Listing 17. Funkcja main()

```
while (1)
{
  if(refresh==1)
  {
    refresh=0;
    switch(stan_gry) //sprawdzanie stanu gry
    {
      case GRA: //gra uruchomiona
        RysujEkran();
        break;
      case GAMEOVER: //koniec gry
        break;
      case PAUZA: //pauza
        break;
      case INTRO: //intro
        break;
    }
  }
}
```

Listing 18. Obsługa przyciśnięcia joysticka
```
switch(stan_gry)
{
  case GRA:  //Jeżeli gra włączona
    stan_gry=PAUZA;  //włącz pauzę
    break;
  case PAUZA:  //Jeżeli gra w czasie pauzy
    stan_gry=GRA;  //wróć do gry
    break;
  case INTRO:  //Jeżeli włączone intro
    stan_gry=GRA;
    break;
  case GAMEOVER:  //Jeżeli nastąpiła przegrana
    CzyscPlansze();  //wyczyść planszę
    KlocekNowy();  //zmień klocek na nowy
    punkty=0;  //wyzeruj licznik punktów
    linie=0;  //i zaliczonych linii
    stan_gry=INTRO;  //przejdź do intro
    break;
}
OdswiezEkran();
```

powinna z czasem przyśpieszać. Proponuję ustawić timer, aby działał 10 razy na sekundę, a w zależności od prędkości gry odliczać określony czas do kolejnego opuszczenia klocka. Do odliczania czasu i ustalania „prędkości" potrzebujemy nowych zmiennych:

```
unsigned char szybkosc;
unsigned char stimer;
```

A odliczanie czasu powierzmy nowej funkcji – **listing 19**. Funkcja wywoływana cyklicznie będzie odmierzała nam czas i zwróci wartość 1, jeżeli przyjdzie pora na kolejne opuszczenie klocka w dół. Teraz wystarczy dodać w funkcji obsługi timera SysTick_Handler:

```
//Funkcja SpeedTimer zwraca 1 co ustaloną
ilość wywołań
  if(TimerGry()==1)
//częstość wywołań reguluje zmienna speed
  {
//Jeżeli trwa gra to opuść klocek i odśwież
ekran
    if(stan_gry==GRA) KlocekDol();
    OdswiezEkran()
  }
```

Klocek będzie spadał z prędkością zależną od wartości zmiennej szybkosc, przy czym im większa wartość tej zmiennej tym szybciej będzie spadał klocek. Spróbujmy nadać wartość 10 zmiennej szybkosc, po czym uruchomić poprawiony program.

Działa? Klocek wreszcie ożył i jest teraz ciągnięty bezlitośnie w dół przez wirtualną grawitację. Pozostaje nam dodać wpis określający początkową prędkość gry także w obsłudze wciśnięcia „ok" zaraz po wyzerowaniu liczników gry. Teraz, aby gra przyspieszała w miarę rozwoju rozgrywki należy gdzieś stopniowo zmniejszać zmienną szybkosc. Ja postanowiłem zmniejszać ją po zaliczeniu każdych kolejnych 10 linii. Oczywiście ty możesz wprowadzić inne reguły gry. Tą kwestię pozostawiam Ci jako ćwiczenie.

Pozostało nam już tylko upiększyć naszą grę, dodać jakąś ładną stronę tytułową i napisy świadczące o pauzie i zakończeniu gry.

Listing 19. Realizacja timera gry
```
unsigned char TimerGry(void)
{
  unsigned char next=0;
  stimer++;  //zwiększ zmienną odliczającą czas
  if(stimer>szybkosc)
//jeżeli minie kolejny odcinek czasu zależny od zmiennej szybkość
  {
    stimer=0;
    next=1;
  }
  return next;  //funkcja zwróci wartość 1
}
```

Zacznijmy od najprostszego napisu PAUZA, jaki pojawi się podczas przerwy w grze. Postanowiłem, że dobrze będzie wyglądał napis PAUZA nałożony na dotychczasowy ekran z grą. Jeżeli naciśniemy „ok" joysticka to napis zniknie, a gra będzie toczyła się dalej. Spróbujmy dodać ten efekt do gry.

Najpierw zaprojektowałem napis, który chcę wyświetlić. Powstaje jednak pytanie: jak wyświetlić go pozostawiając zawartość ekranu jako tło? Tego, co mamy już na wyświetlaczu nie musimy oczywiście w żaden sposób odczytywać. Całą zawartość ekranu mamy w buforze. Teraz przyjrzyjmy się, co znajduje się w funkcji, której użyliśmy poprzednio do wyświetlenia tła:

```
void LcdLoadBMP (const unsigned char*
bitmap)
{
  int i;
  for ( i = 0; i < LCD_CACHE_SIZE; i++ )
  {
    LcdCache[i] = bitmap[i];
  }
}
```

Jest bardzo prosta, kopiuje kolejno 504 bajty z tablicy do bufora ekranu. Można ją łatwo przerobić tak żeby nakładała grafikę z tablicy na dotychczasową zawartość ekranu. Wykorzystałem do tego funkcję logiczną OR i stworzyłem nową funkcje o nazwie LcdLoadAddBMP, która „dodaje" grafikę.

W nowej funkcji linia LcdCache[i] = bitmap[i]; jest zamieniona na LcdCache[i] = LcdCache[i]|bitmap[i]; Teraz wystarczy zamienić projekt napisu PAUSE na tablicę, dokładnie tak samo, jak w przypadku tła i wyświetlić ja przy pomocy nowej funkcji.

Naciśniecie przycisku „ok" uruchomi pauze, zmieniając zmienną stangry. Dzieje się to w obsłudze przerwania wywoływanego przez joystick, która kończy się wywołaniem funkcji OdswiezEkran. W konsekwencji zostanie wykonana procedura w funkcji main, w której znajduje się wcześniej przygotowana przez nas struktura case. Właśnie tam musimy umieścić polecenia rysujące napis PAUSE. Wcześniej pisząc procedurę rysującą planszę gry umieściliśmy ją w osobnej funkcji, ponieważ była skomplikowana. Teraz nie ma takiej potrzeby. Wystarczy, że w odpowiedniej gałęzi struktury case dopiszemy dwa polecenia:

```
  ...
  case PAUZA: //pauza
    LcdLoadAddBMP(pauza);  //nałożenie
napisu PAUSE
    LcdUpdate();
    break;
  ...
```

To wszystko! Jednak ja nie jestem zadowolony z takiego efektu (**rysunek 9**). Napis zlał się nam z tłem i przez to jest całkiem nieczytelny. Żeby to poprawić trzeba by otoczyć go wygaszonymi pikselami. Czeka nas dodanie kolejnej funkcji graficznej, tym razem „odejmującej". Przygotowujemy grafikę tła dla naszego napisu. Ja otworzyłem plik z napisem i otoczyłem go warstwą pikseli dookoła. Z efektu naszej pracy generujemy tablicę z danymi, tym razem nazwaną pauza2. Funkcja odejmująca wygląda tak:

```
void LcdLoadSubBMP (const unsigned char*
bitmap)
{
  int i;
  for ( i = 0; i < LCD_CACHE_SIZE; i++ )
  {
```

```
    LcdCache[i] = LcdCache[i]&( 0xFF-
bitmap[i]);
  }
}
```

Tym razem korzystamy wykonujemy operację AND na buforze i odwróconych danych tła napisu. Odejmowanie bajtów obrazka od wartości 0xFF tworzy jego negatyw. Dodajemy przed poleceniem nałożenia napisu, polecenie „odjęcia tła" LcdLoadSubBMP(pauza2); //nałożenie białego tła pod napis PAUSE. Teraz uzyskaliśmy znacznie lepszy efekt (**rysunek 10**). Analogicznie dodamy słynny napis „Game Over" na zakończenie gry. Wystarczą dwie tablice z grafiką tła i napisu oraz dodanie trzech poleceń do funkcji main.

Pozostała nam kwestia strony tytułowej. Tu chciałbym zaproponować coś ciekawszego niż tylko statyczny obrazek. Wymyśliłem prostą animację, napisy tytułowe, wykonane tę samą techniką, co poprzednie napisy, na przesuwającym się tle. Na wstępie, należy tu zwrócić uwagę, że wpisanie jednego bajtu danych do wyświetlacza zmienia stan 8 pikseli usytuowanych pionowo. Z tego względu najłatwiej będzie nam przesuwać tło w poziomie. Jest mi to nawet na rękę, uzyskamy efekt czołówek rodem z komputerów 8-bitowych. Przygotowujemy obrazek z tłem, ale nie standardowych wymiarów 84×48 punktów tylko szerszy, aby można było go przesuwać. Ponadto, początek grafiki powinien ładnie się zgrywać po połączeniu z początkiem. Moje tło jest szerokie na 126 pikseli i wygląda jak na **rysunku 11**. Konwertujemy grafikę na tablicę danych identycznie jak w przypadku poprzednich grafik. Jednak w tym przypadku uzyskamy większy rozmiar tablicy, ze względu na zwiększony rozmiar grafiki. Warto przy okazji zadeklarować nietypową szerokość obrazka, ułatwi nam to eksperymenty z różnymi tłami:

```
  #define intro_tlo_sz 126
  const unsigned char intro_tlo[intro_tlo_
sz*6]={
    0x00,0x00,0x00,0x00,0x00,0x00, ...
```

Potrzebujemy także dodatkowej zmiennej, w której będzie przechowywane aktualne przesunięcie tła. Deklarujemy ją jako zmienną globalną w pliku main unsigned char introbgoff=0;.

Do biblioteki z funkcjami graficznymi dodajemy trzecią, dodatkową funkcję, tym razem rysującą przesunięte tło. Tym razem jesteśmy zmuszeni operować dwoma współrzędnymi x i y, dlatego użyjemy dwóch pętli. Pierwsze będzie odliczać kolejnych 6 wierszy, a druga 84 bajty każdego wiersza. Przy czym do indeksowania tablicy z grafiką nie użyjemy zmiennej y, tylko dodatkowej zmiennej i do której wpiszemy na początku żądane przesunięcie. Kompletną funkcję pokazano na **listingu 20**. Jeżeli zmienna i osiągnie wartość szerokości obrazka tła, wpisywana jest do niej wartość zero. Dzięki temu obrazek połączy się końcem ze swoim początkiem i będzie się wydawało, że tło nie ma końca.

Kolejną kwestią wymagającą wyjaśnienia jest bardziej skomplikowane indeksowanie tablic. Użyty przeze mnie sposób pozwala indeksować tablicę jednowymiarową, tak jakby była dwuwymiarowa. Zapis:

```
  tablica[x+(y*szerokosc_tablicy)]
```

jest równoważny z zapisem:

```
  tablica[x][y]
```

Oczywiście, jeżeli w tym drugim przypadku tablica zostanie zadeklarowana jako dwuwymiarowa. Nasza nowa funkcja będzie wywoływana z głównej pętli w funkcji main. Należy ją tam umieścić analogicznie do pozostałych funkcji rysujących w następującej postaci

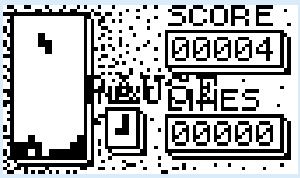
```
LcdLoadBG(intro_tlo,
intro_tlo_sz, introbgoff);
```

Aby tło się zaczęło przesuwać należy jeszcze cyklicznie zwiększać wartość zmiennej introbgoff. Oczywiście najwygodniej będzie to robić w obsłudze przerwania timer SysTick:

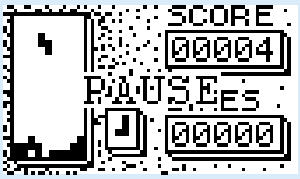
```
  introbgoff++;  //
zwiększenie zmiennej
przesunięcia tła
  if(introbgoff>=126)
introbgoff=0;
```

Dobrana wcześniej częstotliwość wywołań od timera bardzo pasuje do przesuwania tła. Prędkość jest na tyle spokojna, że tworzy to dobry efekt i nie powoduje rozmazywania przesuwanego obrazka. Oczywiście nic nie stoi na przeszkodzie, aby zwiększyć prędkość, z jaką przesuwa się tło, skracając okres wywołań timera SysTick. Należy jednak wtedy pamiętać, aby wyrównać prędkość gry, dobierając odpowiednio początkową wartość zmiennej szybkosc.

Mając już ładnie przesuwające się tło, zaprojektowałem grafiki z napisami tytułowymi. Tak jak uprzednio w przypadku innych „nakładanych grafik" zastosowałem odejmowane tło dla grafiki uzyskując efekt pokazany na **rysunku 12**. W rzeczywistości efekt jest dużo milszy dla oka. Przesuwające się tło, wyraźniej oddziela się od umieszczonych na nim napisów. Pozostało nam zadbać, aby zmienna stan_gry zaraz po uruchomieniu miała wartość INTRO tak, aby program zawsze rozpoczynał się od naszej efektownej czołówki.

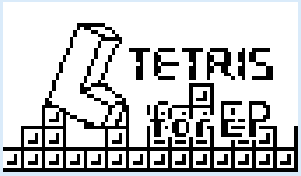


Rysunek 9. Wyświetlenie komunikatu PAUZA za pomocą funkcji sumy logicznej



Rysunek 10. Efekt działania zmodyfikowanej funkcji wyświetlającej komunikaty

Rysunek 11. Wygląd tła gry



Rysunek 12. Ostateczny wygląd „czołówki" programu

Podsumowanie

W tym momencie możemy mówić o końcu „pisania" naszej gry. Z pewnością część Czytelników ma kilka pomysłów na rozbudowę bądź ulepszenie programu. Jeżeli więcej osób będzie zainteresowana np. wykorzystaniem kolorowego wyświetlacza tak, aby każdy spadający klocek był w innym kolorze. To chętnie opiszę takowe modyfikacje na łamach czasopisma. Tym czasem pozdrawiam wszystkich, którzy dotrwali do końca i gorąco zachęcam do modyfikacji opisanej gry.

Rafał Kędzierski

Listing 20. Wyświetlanie bitmapy

```
void LcdLoadBG (const unsigned char* bitmap, unsigned char width,
unsigned char offset)
{
  unsigned char x,y,i;
  for(y=0;y<6;y++)  //Odliczanie wierszy
  {
    i=offset;  //Załadowanie przesunięcia do zmiennej i
    for ( x = 0; x < 84; x++ )  //Odliczanie kolejnych bajtów
    {
      LcdCache[x+(y*84)] = bitmap[i+(y*width)]; //kopiowanie danych
      i++;  //zwiększenie zmiennej i
      if(i>=width) i=0;  //jeżeli zmienna większa od szerokości
tła,
    }  // to wpisz zero
  }
}
```