Mikrokontrolery z rdzeniem ARM, część 13

Porty GPIO

W poprzednich odcinkach zajmowaliśmy się układami peryferyjnymi mającymi bezpośredni wpływ na pracę rdzenia mikrokontrolera. Omówiliśmy także przykładowy plik startowy konfigurujący powyższe układy oraz inicjalizujący pamięć mikrokontrolera zgodnie ze standardem ANSI C/C++.
Tematem bieżącego odcinka będą porty wejścia–wyjścia (GPIO) mikrokontrolerów LPC213x, które umożliwiają bezpośrednie sterowanie układami podłączonymi do wyprowadzeń mikrokontrolera.

Program drugi – wyświetlacz LCD

Kolejnym programem jaki napiszemy w ramach ćwiczeń z portami GPIO będą procedury obsługi znakowego wyświetlacza LCD (HD44180). Procedury te będziemy intensywnie wykorzystywać w dalszej części kursu. W różnych czasopismach o tematyce elektronicznej obsługa znakowego wyświetlacza LCD była poruszana wielokrotnie, dlatego aby nie powielać tych samych schematów tym razem biblioteka ta zostanie napisana w nieco odmienny sposób za pomocą programowania obiektowego C++. W zestawie ZL6ARM linie D0.D7 LCD podłączone są do portu P1.16…P1.23. Linia E jest podłączona do portu P0.30, natomiast RS do portu P0.31. W zestawie niestety nie przewidziano możliwości sterowania linią R/W przez co niemożliwe jest odczytywanie stanu wyświetlacza, dlatego po wysłaniu każdego znaku i rozkazu musimy odczekać pewien czas tak aby wybrana operacja została wykonana. Prawie wszystkie komendy wykonywane są w czasie do 120 μs poza rozkazem czyszczenia wyświetlacza, który może zająć maksymalnie 4,8 ms. Za obsługę LCD odpowiedzialna jest klasa *CLcdDisp*, której deklaracja znajduje się w pliku *CLcdDisp.h*, natomiast definicja została umieszczona w pliku *CLcdDisp.c*. Metody (funkcje) i obiekty (zmienne) zadeklarowane z modyfikatorem *private* mogą być używane tylko wewnątrz klasy, co zapewnia ukrycie ich przed użytkownikiem końcowym. W sekcji tej zapisano stałe związane z wyświetlaczem LCD, takie jak przypisanie bitów odpowiedzialnych za linie *E* i *RW* wyświetlacza oraz stałe związane z komendami kontrolera LCD.

```
//Funkcja opozniajaca
void Delay(unsigned int del);
//Wysyla do portu
void PortSend(unsigned char
data,bool cmd=false);
//Pin E P0.30
static const unsigned int E =
0x40000000;
//Pin RW P0.31
static const unsigned int RS =
0x80000000;
//Maska danych
static const unsigned int DMASK =
0x00FF0000;
//Domyslne sprzetowe
static const unsigned int DELAY_HW
= 15;
//Opoznienie komend
static const unsigned int DELAY_CMD
= 3000;
//Opoznienie dla CLS
static const unsigned int DELAY_CLS
= 30000;
//Komendy wyswietlacza
enum {CLS_CMD=0x01,HOME_
CMD=0x02,MODE_CMD=0x04,ON_CMD=0x08,
SHIFT_CMD=0x10,FUNC_CMD=0x20,CGA_
CMD=0x40,DDA_CMD=0x80};
//Komenda MODE
enum {MODE_R=0x02,MODE_L=0,MODE_
MOVE=0x01};
//Komenda SHIFT
enum {SHIFT_DISP=0x08,SHIFT_
R=0x04,SHIFT_L=0};
//Komenda FUNC
enum {FUNC_8b=0x10,FUNC_4b=0,FUNC_
2L=0x08,
FUNC1L=0,FUNC_5x10=0x4,FUNCx7=0};
};
```

Umieszczono tu także dwie metody: *Delay*, odpowiedzialną za generowanie opóźnień oraz *PortSend* wysyłającą bajt danych do wyświetlacza *Lcd*. Pętla opóźniająca została napisana w asemblerze, aby było możliwe dokładne określenie czasu jej wykonania. Jako argument metody podajemy liczbę, która następnie jest ładowana do któregoś z rejestrów ogólnego przeznaczenia w którym następuje cykliczne odejmowanie liczby jeden, aż do momentu, gdy rejestr ten osiągnie wartość 0.

```
void CLcdDisp::Delay(unsigned int
del)
{
  asm volatile
  (
  "dloop%=:"
  "subs %[del],%[del],#1\t\n"
  "bne dloop%=\t\n"
  : :[del]"r"(del)
  );
}
```

Metoda *PortSend* służy do wysyłania pojedynczego bajtu danych do wyświetlacza LCD. Została ona zadeklarowana następująco:

```
void PortSend(unsigned char data,bo-
ol cmd=false);
```

Jako parametr data przekazujemy instrukcję lub daną, którą chcemy wysłać do wyświetlacza LCD. Gdy parametr *cmd* przyjmie wartość *false*, oznacza to, że liczba przekazana jako data będzie zinterpretowana jako znak do wyświetlenia, w przeciwnym przypadku przesłana dana będzie stanowić rozkaz. W języku C++ możemy deklarować metody i funkcje z parametrami domyślnymi. W przypadku, gdy wywołamy funkcję bez drugiego argumentu parametr cmd przyjmie wartość false, natomiast gdy drugi parametr będzie określony podczas wywołania, argument domyślny będzie ignorowany. Mechanizm ten został stworzony w celu zastąpienia funkcji ze zmienną listą argumentów *(…)* znaną z języka C. Pozwala on zapewnić większą kontrolę nad przekazywanymi argumentami. Działanie tej metody jest następujące. Najpierw sygnał E jest ustawiany w stan 0, w efekcie czego wyświetlacz ignoruje wszystkie stany pojawiające się na liniach danych wyświetlacza. Linie D0..D7 wyświetlacza LCD są zerowane poprzez ustawienie bitów 16:23 w rejestrze IO1CLR. Do portu IO1SET przesyłana jest zawartość zmiennej data przesuniętej o 16 bitów w lewo. W wyniku tych dwóch operacji linie P1.16..P1.23 przyjmują wartość zgodną z zawartością zmiennej *data* bez zmiany pozostałych bitów portu.

```
//E=0
LCDCCLR = E;
//Data = 0;
LCDDCLR = DMASK;
//Wyslij dane
LCDDSET = ((unsigned int)data) <<
16;
```

Po przesłaniu danych na linie D0... D7 następuje ustawienie linii RS w odpowiedni stan w zależności od tego, czy dane przesłane na magistrale zinterpretowane zostaną jako rozkaz (stan wysoki), albo znak do wyświetlenia (stan niski)

```
//Skasuj lub ustaw RS
  if(cmd) LCDCCLR = RS;
  else LCDCSET = RS;
```

Następnie na linii E generowany jest dodatni impuls, w wyniku którego następuje zapisanie danych lub instrukcji do wyświetlacza LCD.

```
//Ustaw Enable
   LCDCSET = E;
   Delay(DELAY_HW);
   //Skasuje enable
   LCDCCLR = E;
```

Wszystkie metody zadeklarowane jako **public** dostępne są dla użytkownika i stanowią zewnętrzny interfejs klasy. Klasa *CLcdDisp* zawiera następujące składowe publiczne:

```
public:
   CLcdDisp();
   ~CLcdDisp();
   void Write(const char *str);
   void Write(char zn);
   void Write(unsigned int licz);
   //Wyczysc wyswietlacz
   void Clear(void);
   //Zalacz wylacz kursor
   void SetCursor(unsigned char cmd);
   void GotoXY(unsigned char
x,unsigned char y);
   template<class T> CLcdDisp& opera-
tor <<(T obj)
   {
      Write(obj);
           return *this;
   }
   CLcdDisp& operator <<(pos obj)
   {
      GotoXY(obj.mx,obj.my);
      return *this;
   }
```

CLcdDisp jest domyślnym konstruktorem klasy i jest on wywoływany podczas tworzenia nowego obiektu danej klasy. W konstruktorze napisano procedurę inicjalizacji wyświetlacza LCD. Inicjalizacja rozpoczyna się od ustawienia linii *RS, E* i *D0...D7* oraz odczekania kilkudziesięciu milisekund na ustabilizowanie napięcia zasilającego:

```
//Konstruktor klasy obslugi wyswie-
tlacza LCD
CLcdDisp::CLcdDisp()
{
   //Linie E i RS jako wyjsciowe
   LCDCDIR |= E|RS;
   LCDCCLR = E|RS;
   //Linia danych jako wyjsciowa
   LCDDDIR |= DMASK;
   Delay(100000);
```

Następnie trzykrotnie wysyłana jest komenda ustawiająca wyświetlacz w tryb 8-bitowy:

```
PortSend(FUNC_CMD|FUNC_8b,true);
Delay(DELAY_CLS);
   PortSend(FUNC_CMD|FUNC_8b,true);
   Delay(DELAY_CMD);
   PortSend(FUNC_CMD|FUNC_8b,true);
   Delay(DELAY_CMD);
```

po czym następuje ustawienie wyświetlacza tak, aby pracował w rozdzielczości 5x7, załączenie wyświetlacza, wyczyszczenie oraz ustawienie kursora w pozycji początkowej. Kolejnymi metodami publicznymi są metody *Write* służące do wypisania na wyświetlaczu pojedynczego znaku, łańcucha tekstowego, oraz liczby stałoprzecinkowej. Uważnego Czytelnika może zdziwić fakt, że metody o takiej samej nazwie zadeklarowane są kilkukrotnie. Jest to kolejna zaleta języka C++, w którym możemy deklarować funkcję i metody o takich samych nazwach. Kompilator w zależności od argumentu przekazanego do metody wywoła odpowiednią funkcję *Write*. Np. jeżeli napiszemy *lcd.Write(100)*, zostanie wywołana metoda *Write*, której argument jest typu *int*. Poszczególne metody są bardzo podobne, przedstawię tutaj metodę *Write* wypisująca łańcuch tekstowy.

```
void CLcdDisp::Write(const char
*str)
{
   while(*str)
   {
     PortSend(*str++);
     Delay(DELAY_CMD);
   }
}
```

Działanie tej metody polega na odczytaniu pojedynczego znaku, przepisaniu jego zawartości do wyświetlacza LCD za pomocą metody *PortSend* oraz odczekaniu około 40 µs na przesłanie znaku. Następnie wskaźnik jest zwiększany o jeden i wysyłany jest kolejny znak. Dzieje się tak do czasu, gdy zostanie wykryty znak 0 będący symbolem końca łańcucha. Metoda *Clear()* umożliwia wyczyszczenie zawartości wyświetlacza, natomiast metoda *GotoXY()* umożliwia przejście do wybranej pozycji kursora. W języku C++ możemy zmieniać znaczenie operatorów, co nosi nazwę przeciążania operatorów. Korzystając z tej techniki napiszemy własne wersje operatora << umożliwiające wypisywanie liczb i zmiennych na przykład tak: `lcd << „Zmienna= „ << zm;` Napiszemy także bardzo prostą klasę *pos*, której przekazanie do obiektu klasy wyświetlacza LCD spowoduje przesunięcie kursora na wybraną pozycję np. tak: `lcd << pos(1,2) << „2 linia”;` Wszystkie operatory korzystają z wcześniej zdefiniowanych metod *Write()* oraz *GotoXY()* i są zdefiniowane w deklaracji klasy zapewniając rozwinięcie ich w miejscu wywołania. Operator wysyłający dane do strumienia zdefiniowano w sposób następujący:

```
template<class T> CLcdDisp& operator
<<(const T &obj)
   {
     Write(obj);
          return *this;
   }
```

Zastosowano tutaj kolejną cechę języka C++ mianowicie funkcję wzorcową. Mechanizm ten umożliwia zadeklarowanie tylko jednej funkcji niezależnie od argumentów jakie ona przyjmuje. Po prostu w momencie wywołania funkcji z danym parametrem, kompilator na etapie kompilacji tworzy daną funkcję zamieniając T na konkretny typ danych na przykład *int*. W wyniku tej czynności nie musimy pisać trzech osobnych wersji operatora dla każdego typu danych: *char*, int, char*. Operator zwraca wskaźnik do klasy obiektu LCD, co umożliwia tworzenie operacji łańcuchowych. W programie stworzono także dodatkową klasę *pos*, której przesłanie do klasy wyświetlacza LCD spowoduje ustawienie kursora na wybranej pozycji. Definicja tej klasy jest następująca:

```
class pos
{
public:
   pos(unsigned char x,unsigned char
y):mx(x),my(y) {}
   unsigned char mx,my;
};
```

Klasa ta zawiera tylko dwa pola określające pozycję kursora na wyświetlaczu oraz konstruktor, który przyjmuje jako argumenty pozycję kursora oraz przepisuje je do *mx* oraz *my*.

Dla obiektu klasy pos stworzony jest osobny operator <<, który wywołuje metodę GotoXY() przesuwając kursor wyświetlacza LCD do odpowiedniej pozycji zawartej w zmiennych *mx*, *my*.

```
CLcdDisp& operator <<(const pos
&obj)
   {
      GotoXY(obj.mx,obj.my);
      return *this;
   }
```

W pliku *testlcd.cpp* znajduje się bardzo prosty programik korzystający z klasy *CLcdDisp*, wypisujący na wyświetlaczu LCD stan wciśniętego klawisza S1..S4.

```
CLcdDisp cout;

//Funkcja glowna main
int main(void)
{
   cout << "Witaj !";
   cout << pos(1,2) << "IO0PIN=";
   unsigned int sk;
   while(1)
   {
      sk = (~IO0PIN >> 4) & 0x0f;
      cout << pos(8,2)<< sk << „ „;
   }
}
```

Działanie programu rozpoczyna się od utworzenia obiektu klasy *CLcdDisp* o nazwie *cout*. W funkcji *main()* wypisywany jest napis powitalny, a następnie program wchodzi w pętlę nieskończoną, która odczytuje stan klawiszy S1...S4 oraz przepisuje ich zawartość do zmiennej *sk*, maskując pozostałe nie istotne bity. Następnie na pozycji 8,2 wypisywany jest stan zmiennej *sk*. Pomimo, że mechanizmy tworzące operatory są trochę zawiłe, korzystanie z samej biblioteki obsługi wyświetlacza LCD jest bardzo proste. Czytelnikom znającym język C++ proponuję napisanie klasy o nazwie *clear*, której przekazanie do klasy *CLcdDisp* za pomocą operatora >> spowoduje wyczyszczenie wyświetlacza LCD.

Lucjan Bryndza, EP
lucjan.bryndza@ep.com.pl