Mikrokontrolery z rdzeniem ARM, część 12

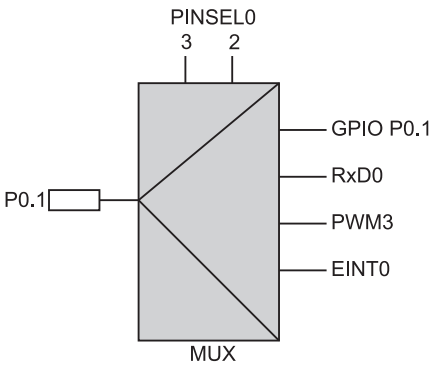
Porty GPIO

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.

Budowa portów GPIO

Mikrokontrolery LPC213x posiadają dwa 32–bitowe porty wejścia/wyjścia P0 i P1, przy czym port P1 ma wyprowadzone tylko najstarsze 16 bitów (P1.16…P1.31). Z portu P0 nie ma wyprowadzonej linii P0.24, natomiast port P0.31 może pełnić tylko funkcję wyjścia. Porty P0 i P1 są dwukierunkowe i mają maksymalną wydajność prądową rzędu 45 mA zarówno od plusa jak i minusa napięcia zasilającego. W przypadku, gdy linie portu skonfigurowane są jako wejściowe, port P0 nie posiada rezystorów podciągających, natomiast port P1 wyposażony jest w rezystory podciągające o wartości 60…300 kΩ. Każdy z pinów może pełnić również rolę jednej z trzech funkcji alternatywnych zapewniając podłączenie wewnętrznych układów peryferyjnych na przykład wyprowadzenie przetwornika A/C. Podobnie jest w innych mikrokontrolerach, jak AVR czy 8051, gdzie każdy port wejścia/ wyjścia może również pełnić rolę wyprowadzenia wewnętrznego układu peryferyjnego, jednak zazwyczaj jest to pojedyncza funkcja. Domyślnie po zerowaniu mikrokontrolera wszystkie piny pełnią rolę portów I/O i są skonfigurowane w kierunku wejściowym. Za rolę, jaką pełni dane wyprowadzenie mikrokontrolera odpowiedzialne są rejestry *PINSELx*. Port P0 posiada dwa rejestry konfiguracyjne *PINSEL0 (0xE002 C000)* oraz *PINSEL1 (0xE002 C004)* natomiast port *P1* z uwagi że ma wyprowadzonych tylko 16 najstarszych bitów posiada, jeden rejestr konfiguracyjny *PINSEL2 (0xE002 C014)*. Do konfiguracji każdego bitu portu P0 wykorzystywane są dwa bity z rejestru PINSELx. Na **rys. 29** przedstawiono budowę jednej linii portu P0 (P0.1).



Rys. 29.

W zależności od stanu bitów [3..2] rejestru *PINSEL0* port linia P0.1 mikrokontrolera pełni rolę portu wejścia/wyjścia (00b), wejścia RxD pierwszego portu szeregowego (01b), wyjścia PWM (10b), lub wejścia przerwania zewnętrznego (11b) Zastosowanie dodatkowych rejestrów oraz multipleksera wyboru funkcji alternatywnej jest bardzo interesującym rozwiązaniem, ponieważ nie musimy konfigurować portów wejścia/wyjścia w odpowiednim kierunku. Wybranie funkcji alternatywnej spowoduje automatyczne ustawienie linii portu w kierunku odpowiadającym pełnionej funkcji. W **tab. 17** przedstawiono wszystkie funkcje, jakie mogą pełnić poszczególne linie portu P0 wraz z odpowiednią kombinacją bitów rejestrów *PINSEL0* oraz *PINSEL1* potrzebną do ustawienia odpowiedniej funkcji.

W przypadku portu P1 sytuacja jest dużo prostsza, ponieważ jedyną funkcją alternatywną jaką pełni ten port, jest interfejs debugowania i śledzenia, którego raczej w praktyce amatorskiej nie będziemy wykorzystywać. Za sterowanie funkcjami alternatywnymi portu P1 odpowiedzialny jest rejestr PINSEL2, który zawiera bity konfiguracyjne pokazane w **tab. 18**.

Po wyzerowaniu mikrokontrolera badany jest stan linii P1.26 i P1.20. W przypadku, gdy linia P1.26 podczas zerowania znajdzie się w stanie niskim wówczas interfejs DEBUG jest włączany. Natomiast, gdy linia P1.20 podczas zerowania będzie się znajdować w stanie niskim, wówczas włączony będzie interfejs TRACE. Ponieważ linie portu P1 posiadają rezystory podciągające, pozostawienie ich nie podłączonych spowoduje że domyślnie po zerowaniu interfejs DEBUG i TRACE będzie wyłączony. Linie mikrokontrolera po zerowaniu domyślnie pracują jako porty I/O więc gdy chcemy skorzystać z wybranych funkcji alternatywnych portu musimy odpowiednio skonfigurować go poprzez zapis odpowiednich wartości do rejestrów PINSELx. Na przykład, jeżeli chcemy skorzystać z portu szeregowego UART0, który wykorzystuje linie RxD0 i TxD0, należy ustawić odpowiednie bity w rejestrze PINSEL0:

```
#define PINTXD0 0x01
#define PINRXD0 (0x01<<2)

PINSEL0 |= PINTXD0 | PINRXD0; //Linie
P0.0 I P0.1 jako TxD0 i RxD0
```

Porty mikrokontrolerów LPC213x w przeciwieństwie do mikrokontro-

Tab. 17. Funkcje pełnione przez poszczególne linie portu P0

Port	Rejestr PINSELx	00b	01b	10b	11b
P0.0	PINSEL0 [1:0]	P0.0	TxD0	PWM1	–
P0.1	PINSEL0 [3:2]	P0.1	RxD0	PWM3	EINT0
P0.2	PINSEL0 [5:4]	P0.2	SCL0	CAP0.0	–
P0.3	PINSEL0 [7:6]	P0.3	SDA0	MAT0.0	EINT1
P0.4	PINSEL0 [9:8]	P0.4	SCK0	CAP0.1	AD0.6
P0.5	PINSEL0 [11:10]	P0.5	MISO0	MAT0.1	AD0.7
P0.6	PINSEL0 [13:12]	P0.6	MOSI0	CAP0.2	AD1.0
P0.7	PINSEL0 [15:14]	P0.7	SSEL0	PWM2	EINT2
P0.8	PINSEL0 [17:16]	P0.8	TxD1	PWM4	–
P0.9	PINSEL0 [19:18]	P0.9	RxD1	PWM6	EINT3
P0.10	PINSEL0 [21:20]	P0.10	RTS1	CAP1.0	AD1.2
P0.11	PINSEL0 [23:22]	P0.11	CTS1	CAP1.1	SCL1
P0.12	PINSEL0 [25:24]	P0.12	DSR1	MAT1.0	AD1.3
P0.13	PINSEL0 [27:26]	P0.13	DTR1	MAT1.1	AD1.4
P0.14	PINSEL0 [29:28]	P0.14	DCD1	EINT1	SDA1
P0.15	PINSEL0 [31:30]	P0.15	RI1	EINT2	AD1.5
P0.16	PINSEL1 [1:0]	P0.16	EINT0	MAT0.2	CAP0.2
P0.17	PINSEL1 [3:2]	P0.17	CAP1.2	SCK	MAT1.2
P0.18	PINSEL1 [5:4]	P0.18	CAP1.3	MISO	MAT1.3
P0.19	PINSEL1 [7:6]	P0.19	MAT1.2	MOSI	CAP1.2
P0.20	PINSEL1 [9:8]	P0.20	MAT1.3	SSEL	EINT3
P0.21	PINSEL1 [11:10]	P0.21	PWM5	AD1.6	CAP1.3
P0.22	PINSEL1 [13:12]	P0.22	AD1.7	CAP0.0	MAT0.0
P0.23	PINSEL1 [15:14]	P0.23	–	–	–
P0.24	PINSEL1 [17:16]	–	–	–	–
P0.25	PINSEL1 [19:18]	P0.25	AD0.4	AOUT	–
P0.26	PINSEL1 [21:20]	P0.26	AD0.5	–	–
P0.27	PINSEL1 [23:22]	P0.27	AD0.0	CAP0.1	MAT0.1
P0.28	PINSEL1 [25:24]	P0.28	AD0.1	CAP0.2	MAT0.2
P0.29	PINSEL1 [27:26]	P0.29	AD0.2	CAP0.3	MAT0.3
P0.30	PINSEL1 [29:28]	P0.30	AD0.3	EINT3	CAP0.0
P0.31	PINSEL1 [31:30]	P0.31 (Out)	–	–	–

lerów rodziny 51 są w pełni dwukierunkowe. Do sterowania portami służą następujące rejestry SFR:

Rejestr kierunku *IO0DIR* *(0xE00028008)* (port P0), *IO1DIR (0xE00028018)* (port P1) umożliwia wybór kierunku pracy wybranej linii I/O. Ustawienie bitu w tym rejestrze powoduje, że odpowiadająca mu linia I/O pełni rolę wyjścia, natomiast jego wyzerowanie powoduje, że wybrana linia pełni rolę wejścia. Na przykład wykonanie operacji *IO0DIR=0x02;* spowoduje ustawienie linii P0.1 jako wyjściowej.

Rejestr *IO0PIN* *(0xE0028000)* oraz *IO1PIN (0xE0028010)* umożliwia odczytanie oraz ustawienie stanu wybranej linii I/O. W przypadku, gdy wybrany pin skonfigurowany jest jako wejściowy odczyt tego rejestru jest bezpośrednim odzwierciedleniem stanu sygnałów elektrycznych panujących na tym pinie, natomiast, gdy wybrana linia skonfigurowana jest jako wyjściowa, odczytanie tego rejestru powoduje odczytanie stanu wewnętrznych przerzutników portu i odzwierciedla stan w jakim znajduje się wybrana linia wyjściowa. Zapis do tego rejestru w przypadku gdy wybrana linia skonfigurowana jest jako wyjściowa powoduje wystawienie stanów logicznych odzwierciedlających stan rejestru na odpowiednich pinach mikrokontrolera. Na przykład IO0PIN=0, spowoduje ustawienie wszystkich linii portu P0 w stan niski.

Rejestr *IO0SET* (*0xE00028004*) oraz *IO1SET* (*0xE00028014*) umożliwia ustawienie wybranych linii I/O w stan wysoki („1") bez zmiany stanu pozostałych linii. Na przykład instrukcja IO0SET=0x80 spowoduje ustawienie P0.7 w stan wysoki bez zmiany stanu pozostałych linii.

Rejestr *IO0CLR* (*0xE00028004*) oraz *IO1CLR* (*0xE00028014*) umożliwia ustawienie wybranych linii I/O w stan niski („0") bez zmiany stanu pozostałych linii. Wpisanie 1 na wybranym bicie powoduje wyzerowanie odpowiadającego bitu w porcie I/O. Na przykład instrukcja IO0CLR=0x80 spowoduje ustawienie P0.7 w stan niski bez zmiany stanu pozostałych linii.

Jak więc widzimy sterowanie portami I/O mikrokontrolera jest bardzo proste. Aby odczytać zawartość linii wejścia/wyjścia mikrokontrolera, wystarczy skonfigurować wybraną linię jako wejściową za pomocą rejestru kierunku IOxDIR, a następnie odczytać stan wybranej linii z rejestru IOxPIN. Natomiast jeżeli chcemy ustawić wybrane linie w odpowiedni stan wystarczy za pomocą rejestru IOxDIR ustawić wybrane linie jako wyjściowe i za pośrednictwem par rejestrów IOxSET, IOxCLR lub IOxPIN ustawić odpowiednie bity. Zastosowanie rejestrów IOxSET i IOxCLR jest bardzo wygodne ponieważ możemy ustawić lub skasować wybrane bity portu bez wcześniejszego ich odczytywania. W przypadku, gdy chcemy zmienić całą zawartość danego portu, wygodniej będzie skorzystać z rejestru IOxPIN, który od razu ustawi cały port zgodnie z zawartością rejestru. Wszystko wygląda bardzo kolorowo, jednak jest jeden drobny mankament charakterystyczny dla mikrokontrolerów LPC213x. Mianowicie rejestry wejścia/wyjścia są umieszczone w obszarze rejestrów VPB do których dostęp odbywa się za pomocą stosunkowo wolnej magistrali urządzeń peryferyjnych VPB, w wyniku czego rdzeń potrzebuje dodatkowych cykli, aby przesłać zawartość tego obszaru pamięci do rejestru ogólnego przeznaczenia. Efektem tego jest bardzo wolny dostęp do porów I/O mikrokontrolera. Dla porównania AVR z zegarem 16 MHz potrafi szybciej

Tab. 18.

Bit	Nazwa	Opis	Wart. pocz.
[1:0]	–	Zarezerwowane	0
[2]	GPIO/DEBUG	0 – Linie P1.26..P1.31 pracują jako porty IO 1 – Linie P1.26..P1.31 skonfigurowane są jako port DEBUG	~P1.26
[3]	GPIO/TRACE	0 – Linie P1.25..P1.16 pracują jako porty IO 1 – Linie P1.25..P1.16 skonfigurowane są jako port DEBUG	~P1.20
[31:4]	–	Zarezerwowane	–

zmieniać stan linii portu I/O niż LPC213x pracujący z częstotliwością 60 MHz. Konstruktorzy Philipsa szybko zauważyli ten problem i w mikrokontrolerach LPC214x obszar portów wejścia/wyjścia został podłączony bezpośrednio do magistrali lokalnej i porty te zostały nazwane szybkimi portami GPIO. W mikrokontrolerach LPC214x rejestry te nie zostały bezpośrednio przeniesione pod nowy obszar, tylko dodano nowy zestaw rejestrów, a stare rejestry dla kompatybilności wstecznej nadal znajdują się na swoim miejscu. Sposób korzystania z tych rejestrów zostanie przedstawiony w ostatnim odcinku cyklu, który będzie poświecony w całości nowemu LPC214x.

Trochę praktyki – przykładowy program

Mając już odpowiednią dawkę wiedzy teoretycznej zajmiemy się teraz napisaniem prostego programu mającego na celu zapoznanie się z rejestrami portów GPIO. Oczywiście i w tym przypadku będziemy korzystać z zestawu uruchomieniowego ZL6ARM. Działanie programu będzie następujące: po wciśnięciu przycisku S1 zostanie zapalona dioda LED0 oraz wyłączona dioda LED1; po wciśnięciu przycisku S2 dioda LED0 zgaśnie, natomiast dioda LED1 zostanie zapalona. Po wciśnięciu klawisza S3 stan diody LED3 zostanie zmieniony na przeciwny. W programie tym wykorzystamy omawiane we wcześniejszych odcinkach pliki startowe, dlatego nie będziemy się już nimi tutaj zajmować. Przykładowy program można także ściągnąć ze strony EP (*ep5a.zip*) i zaimportować do środowiska *Eclipse*. Pisanie programu rozpoczynamy od zdefiniowania stałych odpowiadających bitom poszczególnych diod, przycisków oraz portów do których są podłączone diody LED i klawisze, co ułatwi późniejsze zmiany oraz wpłynie na większą przejrzystość kodu:

```
#define LEDDIR IO1DIR     //Rejestr kierunku LED
#define LEDSET IO1SET     //Rejestr ustawiający bity LED
#define LEDCLR IO1CLR     //Rejestr kasujący bity LED
#define LEDPIN IO1PIN     //Rejestr portu LED

#define KEYDIR IO0DIR     //Rejestr kierunku klawiszy
#define KEYPIN IO0PIN     //Rejestr portu klawiszy

#define LEDY (0xFF<<16)   //Wszystkie LEDY P.16..P1.24
#define LED0 (1<<16)      //P1.16 - Dioda LED0
#define LED1 (2<<16)      //P1.17 - Dioda LED1
#define LED2 (4<<16)      //P1.18 - Dioda LED2

#define S1 0x10     //P0.4 - Klawisz S1
#define S2 0x20     //P0.5 - Klawisz S2
#define S3 0x40     //P0.6 - Klawisz S3
```

Działanie programu rozpoczyna się w funkcji *main* od ustawienia rejestrów kierunku. Bity portu P1.16...P1.18 odpowiedzialne za sterowanie diodami LED ustawiane są w kierunku wyjściowym, natomiast linie portu do których są podłączone klawisze S1, S2, S3 ustawiane są jako wejściowe:

```
//Kierunek dla ledow wyjście
LEDDIR |= LEDY;
//Kierunek dla klawiszy wejście
KEYDIR &= ~(S1|S2|S3);
```

Operacja ustawienia linii portu P0.4...P0.6 nie są niezbędne, ponieważ po wyzerowaniu mikrokontroler ustawia wszystkie linie I/O w kierunku wejściowym. Tak samo w programie tym nie ustawiamy w ogóle bitów rejestru PINSELx, ponieważ domyślnie po zerowaniu do linii wejściowych podłączone są porty GPIO. Po tej czynności program wchodzi do pętli nieskończonej **while**(1){…}, w której sprawdzane jest wciśnięcie klawisza S1 (stan niski) i w przypadku jego naciśnięcia włączana jest dioda LED0 oraz wyłączana LED1:

```
if(!(KEYPIN & S1))
{
  //!Jezeli wcisniety S1 to zalacz LED0 i wylacz LED1
  LEDSET = LED0;
  LEDCLR = LED1;
}
```

Sprawdzanie wciśnięcia klawisza odbywa się poprzez odczytanie rejestru *IO0PIN (KEYPIN)*, natomiast włączanie i wyłączanie diod odbywa się poprzez wpisanie jedynki na wybranym bicie w rejestrze *IO1SET (LEDSET)* gdy chcemy załączyć wybraną diodę (linia portu przyjmie wówczas stan wysoki) i poprzez wpisanie jedynki na wybranym bicie w rejestrze *IO1CLR (LEDCLR)* gdy chcemy wyłączyć wybraną diodę (linia portu przyjmie wówczas stan niski). Sprawdzanie wciśnięcia stanu klawisza S2 oraz załączenie diody LED1 i wyłączenie LED0 odbywa się w sposób analogiczny jak poprzednio. Dla pokazania sposobu sterowania wyjściami za pomocą rejestru IOxPIN działanie fragmentu programu odpowiedzialnego za wykrycie wciśnięcia klawisza S3 jest trochę inne. Wykrywane jest zbocze opadające na linii klawisza S3 (P0.6) poprzez porównanie bieżącego i poprzedniego stanu klawisza:

```
//Jezeli zbocze opadajace na S3 to zmien stan LED2
key = KEYPIN & S3;
if(pkey && !key)
{
  LEDPIN ^= LED2;
}
pkey = key;
```

W przypadku, gdy zostanie wykryte zbocze, stan linii P1.18 (LED2) jest zmieniany na przeciwny za pomocą operacji XOR na rejestrze LEDPIN (P1PIN).

Uruchamiając ten program możemy zauważyć że nie jest on odporny na drgania styków. Nie ma to znaczenia w przypadku reakcji na klawisz S1 i S2, ponieważ gdy linia danego portu jest już skasowana lub ustawiona, to ponowne skasowanie lub ustawienie tej samej linii nie spowoduje żadnych efektów. Natomiast w przypadku wciśnięcia klawisza S3 stan linii portu zmieniany jest na przeciwny. Możemy więc zauważyć wielokrotne zmiany stanu diody LED2. Modernizację programu tak, aby był odporny na drgania zestyków pozostawiam Czytelnikowi jako ćwiczenie do samodzielnego wykonania.

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

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++. Nie będziemy tutaj szczegółowo omawiać aspektów działania wyświetlacza LCD, a zainteresowanych odsyłam do EdW 11/97. W zestawie ZL6ARM linie D0.D7 LCD podłączone są do portu P1.16...P1.23. Linia E podłączona jest 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 okres czasu 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 linię *E* i *RW* wyświetlacza oraz stałe związane z komendami kontrolera LCD.

Klasy, Obiekty, oraz programowanie zorientowane obiektowo w skrócie

Pisząc w języku C program, który dotyczy jakiś realnych obiektów na przykład regulatora temperatury, tablicy świetlnej, sterownika akwariowego musimy wszelkie zależności i wielkości zamienić na zestaw luźnych liczb i funkcji operujących na danych. Na przykład w zmiennej *float temp* trzymamy temperaturę bieżącą, przy czym tylko my wiemy że jest to temperatura zadana. Równie dobrze liczbę reprezentującą temperaturę moglibyśmy podstawić do jakiejś innej funkcji realizującą całkiem inne zadanie a kompilator nawet nie zaprotestowałby tylko wyliczyłby jakieś bzdury. Natomiast otaczający nas świat nie składa się z luźnych liczb i funkcji tylko z obiektów. Na przykład wspomniany regulator temperatury jest obiektem, który z kolei zawiera w sobie obiekty takie jak wyświetlacz LCD, czujnik temperatury, czy klawiaturę. Właśnie język C++ pozwala nam działać w sposób obiektowy umożliwiając budowanie modeli rzeczywistych obiektów, a nie luźnego zestawu liczb oraz funkcji. Każdy model posiada zestaw danych (pól) oraz zachowań (metod). Na przykład wyświetlacz LCD posiada dane w postaci tekstu do wyświetlenia oraz zachowania (metody) takie jak wyczyszczenie wyświetlacza, wypisanie liczby, czy przesunięcie kursora na wskazaną pozycję. Zbierając te wszystkie dane i zachowania w jedną całość budujemy konkretny typ (klasę) wyświetlacza LCD. Wymyśliliśmy więc opis umożliwiający zbudowanie konkretnego egzemplarza (obiektu) wyświetlacza LCD, nie jest to jeszcze żaden konkretny wyświetlacz. Definicja klasy w języku C++ ma następującą postać:

```
class budowany_typ
{
public:                                    //
Specyfikator dostepu
   budowany_typ();
   //Konstruktor klasy
   ~budowany_typ();
   //Destruktor klasy
   metoda1();

   metoda2();

protected:                                 //Spe-
cyfikator dostepu
   metoda3();

private:                                   //Spe-
cyfikator dostepu
   int pole1;
   float pole2;
};
```

Zdefiniowane własnej klasy nie jest trudne najpierw występuje tutaj słowo kluczowe ***class*** następnie występuje nazwa klasy po czym klamra a w niej ciało klasy. W ciele klasy deklarujemy wszelkie metody (zachowania) i pola (dane) klasy. Jest to bardzo ważny aspekt bowiem w definicji klasy zamknęliśmy wszelkie pola i metody klasy co nazywamy **enkapsulacją** danych. W deklaracji klasy znajdują się także specyfikatory dostępu *public, protected, private*. Etykieta *private* oznacza że pola i metody znajdujące się pod nią dostępne są tylko z wnętrza klasy. Etykieta *protected* oznacza że pola i metody znajdujące się pod nią są dostępne dla klas dziedziczonych od tej klasy. (O dziedziczeniu będziemy jeszcze mówić przy innej okazji) . Natomiast etykieta *public* oznacza że pola i metody dostępne są wewnątrz jak i na zewnątrz klasy. Zastosowanie specyfikatorów dostępu pozwala ukryć przed użytkownikiem końcowym wszelkie mechanizmy wewnętrzne klasy. Po prostu użytkownik korzystający np. z klasy wyświetlacza LCD nie powinien mieć dostępu do metody przesyłającej na magistralę bajt danych, metoda ta powinna być wywoływana tylko przez inne metody z wnętrza klasy. W sekcji *public* widzimy metodę której nazwa jest identyczna jak nazwa klasy jest to tak zwany **konstruktor** klasy, który jest specjalną metodą wywoływaną w momencie tworzenia obiektu danej klasy. Umożliwia nam to wykonanie pewnych czynności zanim obiekt danej klasy powstanie. Np. tworząc obiekt klasy wyświetlacz LCD w konstruktorze będziemy inicjalizować wyświetlacz tak aby był on w stanie wyświetlać znaki. Tworząc na przykład klasę pojemnika na liczby w konstruktorze tej klasy alokować będziemy pamięć do przechowywania tych liczb. W konstruktorze nie ma żadnej magii jest to po prostu zwykła funkcja której osobliwością jest to że jest ona wywoływana w momencie tworzenia obiektu danej klasy. Analogiczną funkcją do konstruktora, wywoływaną w momencie niszczenie obiektu danej klasy jest **destruktor** klasy wywoływany w momencie gdy obiekt danej klasy przestaje istnieć. Destruktor deklarujemy poprzedzając metodę o takiej samej nazwie jak klasa znakiem ~. Na przykład we wspomnianym wcześniej pojemniku na liczby destruktor będzie zawierał funkcje dealokacji pamięci którą wcześniej przydzieliliśmy w konstruktorze. Definicję klasy najczęściej tworzymy w plikach nagłówkowych *.h. Natomiast deklarację poszczególnych metod możemy zawrzeć we wnętrzu definicji ciała samej klasy np.

```
class mojaklasa
{
public:                                    //
Specyfikator dostepu

   int metoda1(int a)
       {
          return a*a + mx;
       }
   int mx;

};
```

Wówczas metoda ta zostanie potraktowana jako metoda *inline* i zostanie rozwinięta w miejscu wywołania. Metody zawierające więcej niż kilka linijek kodu powinny być zadeklarowane w plikach *.c. w sposób następujący.

```
Zwracany_typ Nazwa_klasy::NazwaMetody(ar-
gumenty)
{
       //Ciało metody
}
```

Widzimy że nazwę metody poprzedza nazwa klasy zakończona specyfikatorem dostępu :: co określa że dana metoda należy do danej klasy. Na przykład we wspomnianym wcześniej przykładzie klasy *mojaklasa* zadeklarowanie metody klasy w pliku *.c wygląda następująco:

```
int mojaklasa::metoda1(int a)
{
     return a*a + mx;
}
```

To o czym wcześniej mówiliśmy było tylko definicją klasy określającą sposób w jaki ona była zbudowana. Sama definicja klasy nie deklaruje żadnych obiektów (egzemplarzy) tej klasy. Utworzenie konkretnych obiektów danej klasy odbywa się w taki sam sposób jak tworzenie obiektów typów wbudowanych np. *int a,b,c,d;* spowoduje utworzenie 4 obiektów typu int o nazwach a b c d. Tak samo napisanie *mojaklasa a,b,c,d*; spowoduje utworzenie czterech obiektów o nazwach *a b c d* klasy *mojaklasa*. Należy sobie uzmysłowić że utworzenie 4 obiektów klasy *mojaklasa* spowoduje utworzenie 4 oddzielnych kompletów danych dla poszczególnych obiektów danej klasy. Natomiast metody operujące na tych składnikach definiowane są tylko jednokrotnie. Każda metoda do pól danej klasy odwołuje się za pomocą wskaźnika ***this***, który pokazuje na konkretny egzemplarz danej klasy. Na przykład we wspomnianej wcześniej metodzie metoda1() odwołanie do pola *mx* będącego składnikiem danej klasy odbywa się za pomocą wskaźnika this następująco: return a*a + this->mx. Wskaźnik ten jest tutaj wywoływany niejawnie przez kompilator, ale my czasami będziemy z niego świadomie korzystać. Odwołanie do wybranego pola danej klasy odbywa się za pomocą znaku kropki. Na przykład wpisanie b=a.mx spowoduje przepisanie pola mx obiektu a do zmiennej b. Natomiast wywołanie metod na rzecz konkretnego obiektu odbywa się poprzez wpisanie po kropce danej metody. Na przykład *c.metoda1(4)* spowoduje wywołanie metoda1() działającej na danych będących składowymi obiektu b.

```
//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.

Przeładowanie nazw funkcji i metod

Programując w języku C przyzwyczailiśmy się że w programie może być tylko jedna funkcja o takiej samej nazwie. W języku C++ natomiast może istnieć więcej niż jednak funkcja lub metoda w obrębie klasy posiadająca taką samą nazwę pod warunkiem że posiada ona inną listę argumentów. Inaczej rzecz mówiąc kompilator C++ rozpoznaje funkcje lub metodę nie tylko po samej nazwie ale też po liście argumentów. Na przykład w C gdybyśmy chcieli napisać funkcję do wyświetlania poszczególnych typów danych na wyświetlaczu LCD musielibyśmy dla każdego typu zdefiniować funkcję o innej nazwie: *WriteInt(int w); WriteStr(char *s); WriteChar(char c);* W momencie gdy chcieliśmy wypisać konkretny typ danej na przykład int musieliśmy wywołać funkcję WriteInt(). W języku C++ możemy natomiast zdefiniować trzy funkcje o takiej samej nazwie Write z inną listą argumentów np. tak: *Write(int w); Write(char *s); Write(char c);* W momencie wywołania funkcji nie musimy się zastanawiać którą wersję funkcji chcemy wywołać po prostu piszemy Write(„Text") a kompilator sam na podstawie listy argumentów ustali że trzeba wywołać funkcję *Write(char *s);*

```
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 zinterpretowana będzie jako znak do wyświetlenia, w przeciwnym przypadku przesłana dana stanowić będzie rozkaz. W języku C++ możemy deklarować metody i funkcję 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 ustawiany jest 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 linię 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 kurosa 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 funkcje Write. Np. jeżeli napiszemy lcd. Write(100) wywołana zostanie 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 zwiększany jest 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. Nie będę ich tutaj przedstawiał ponieważ odbywa się to na zasadzie wysłania odpowiedniego kodu komendy oraz odczekania określonego czasu na jej wykonanie. Czytelnicy którzy programowali w języku C++ zapewne korzystali z biblioteki standardowej iostream która umożliwiała wypisywanie komunikatów i zmiennych na ekran poprzez wpisanie danych do obiektu cout Na przykład: `cout << „Zmienna= ” << Zmienna << endl`; Przesyłanie danych do obiektu odbywa się za pomocą operatora <<. W języku C++ możemy zmieniać znaczenie operatorów, co nosi nazwę przeciążania operatorów. Korzystając z tej techniki napiszemy własne wersję 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 ich 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. Działanie operatora << jest bardzo proste mianowicie parametr który otrzymuje operator przekazywany jest do funkcji Write, która wypisuje w odpowiedni sposób dane na wyświetlaczu LCD. Operator zwraca wskaźnik do klasy obiektu LCD co umożliwia tworzenie operacji łańcuchowych. W programie stworzono także dodatkową klasę pos, której przesłanie do

Przeładowanie operatorów

W języku C++ istnieje możliwość zdefiniowania własnych operatorów czyli możemy sprawić żeby znaczki takie jak +,–,*,/ wykonywały dla nas jakieś czynności na rzecz tworzonych przez nas klas. Możemy na przykład sprawić że operator pełniący rolę przesunięcia bitowego w stosunku do wbudowanych typów danych << dla klasy wyświetlacza LCD będzie wypisywał znaki na ekranie. W przypadku wbudowanych typów danych na przykład int, gdy wpiszemy a*b kompilator po prostu wywoła specjalną funkcję powodująca pomnożenie dwóch argumentów a i b. Podobnie stanie się na przykład gdy a i b będą zdefiniowane jako double zostanie wówczas wywołana funkcja mnożenia dwóch liczb typu double. W C++ możemy zdefiniować własne wersje dowolnego operatora które wykonują jakieś czynności na stworzonych przez nas typach danych (klasach). Na przykład gdy mamy obiekty naszaklasa a,b; i napiszemy a*b kompilator wywoła naszą funkcję operatorową *, która wykona jakąś operację na naszym obiekcie. (Oczywiście jeżeli została ona wcześniej zdefiniowana). Operator może być napisany jako oddzielna funkcja lub jako metoda składowa klasy. Na przykład operator dodawania dla własnego typu danych zdefiniowany jako funkcja ma następującą postać:

```
mojtyp operator+(mojtyp a,mojtyp b)
{
  return a+b;
}
```

Taki sam operator dodawania możemy zadeklarować jako metodę składową klasy:

```
mojtyp mojtyp::operator+(mojtyp b)
{
  return this->a + b;
}
```

Widzimy że w tej definicji zniknął jeden argument, ponieważ funkcja jest teraz składową klasy to znaczy że jest wykonywana na rzecz konkretnego obiektu, zatem dostaje do niego wskaźnik this do obiektu który jest właśnie pierwszym argumentem funkcji operatorowej. W ten sposób możemy również przeładowywać inne operatory. Musimy tylko pamiętać że nie możemy zmienić znaczenia operatorów dla typów wbudowanych na przykład int. Bardzo ważną informacją jest również ze priorytety operatorów są zawsze takie same i ściśle określone i nie możemy zmieniać priorytetów operatorów.

Zrozumienie mechanizmu definiowania operatorów dla własnych typów klas będzie łatwiejsze gdy zobaczymy w jaki sposób odbywa się to dla jakiegoś typu wbudowanego. Na przykład gdy mamy zdefiniowane dwie zmienne float a=12; float b=15; i wpiszemy a*b wówczas zostanie wywołana funkcja operator*(a,b) która w gdzieś tam we wnętrzu kompilatora zdefiniowana jest następująco:

```
float operator*(float a,float b)
{
  return a*b;
}
```

Funkcje i metody wzorcowe

Gdybyśmy chcieli zapisać bardzo prosty algorytm wyliczający na przykład minimum musielibyśmy dla każdej pary argumentów (np. int, float, double itd.) stworzyć oddzielne wersje funkcji min(), co niepotrzebnie komplikuje i wydłuża program. W języku C++ istnieje mechanizm funkcji i metod wzorcowych w którym zamiast z góry określać typy argumentów i zwracane wartości, można niektóre lub wszystkie z tych typów zastąpić parametrami, natomiast sama treść funkcji nie zmieni się. Na przykład wspomniana wcześniej funkcja wyliczająca minimum wygląda następująco:

```
template <class Typ> Typ min(Typ a,Typ b)
{
  return a<b ? a : b;
}
```

Definicję funkcji wzorcowej poprzedza słowo kluczowe template po którym następuje lista parametrów formalnych oddzielonych przecinkami. Każdy parametr składa się ze słowa kluczowego class określającym że typem może być zarówno typ wbudowany jaki klasa zdefiniowana przez użytkownika. Zadeklarowany w ten sposób parametr formalny może być używany jak typ wbudowany lub klasa użytkownika w pozostałej części funkcji wzorcowej. Dalsza deklaracja funkcji nie różni się niczym od zwykłych niewzorowych funkcji. W powyższym przykładzie parametr Typ służy do określenia typu wartości przekazywanych do funkcji min oraz wartości zwracanej przez nią. Za każdym razem gdy funkcja min() zostanie użyta w miejsce parametru Typ podstawiony zostanie odpowiedni dla danego przypadku typ wbudowany np. gdy wpiszemy min(10.0,12.0) za parametr Typ zostanie podstawiony typ wbudowany float. Proces prowadzący do podstawienia właściwego typu nazywa się konkretyzowaniem wzorca. Po prostu kompilator na podstawie wzorca sam stworzy sobie odpowiednią wersję funkcji operującą na określonym typie danych w tym przypadku float.

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 pozycje 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