C dla mikrokontrolerów 8051

część 1

Przedstawiamy pierwszą część kursu języka C dla mikrokontrolerów. Zamieszczono w nim przykłady wzięte z praktyki inżynierskiej, w związku z czym ich praktyczna wartość jest bardzo duża.

Przygotowując już od kilkunastu lat programy, dostrzegłem pewien trend, który jest widoczny nie tylko w środowiskach projektowania oprogramowania zarówno dla „dużych" komputerów, jak również dla mikrokontrolerów jednoukładowych.

Języki programowania stają się coraz łatwiejsze do opanowania. Oferują wiele gotowych procedur, których użycie jest bardzo łatwe. Doskonałym przykładem takiego oprogramowania jest Bascom, w którym rzeczy skomplikowane i trudne stają się bardzo łatwe. Bascom jednak ma jedną podstawową wadę - tworzy kod, który jest uniwersalny a przez to nie optymalny, w związku z czym zajmuje dużo pamięci. Oczywiście, nowoczesne mikrokontrolery mają tej pamięci coraz więcej, jednak ma to także inne reperkusje. Sam bardzo często korzystam z takich języków, jak Bascom, zwłaszcza jeśli szybko potrzebuję mieć program, w stosunku do którego nie mam zbyt wielkich wymagań. Jednak stojąc przed zadaniami „poważniejszymi", nieodmiennie korzystam z języka C i asembler'a.

Asembler jest językiem, który zmusza do uczenia się programowania praktycznie od podstaw. Daje krótkie kody wynikowe i umożliwia w zasadzie dowolną optymalizację kodu programu. Jednak każdy, kto kiedykolwiek napisał program w tym języku wie, że bardzo trudno szuka się w nim błędów, a operacje nawet pozornie proste trzeba wykonać używając wielu rozkazów. Jednym słowem - programy pisane w asemblerze pochłaniają wiele czasu, którego nie zawsze mamy w nadmiarze, szczególnie gdy kończymy projekt.

Większości wad asemblera pozbawiony jest język C: zwarty i czytelny kod źródłowy (pod warunkiem, że celowo czegoś w nim nie ukrywamy), bardzo oszczędny kod wynikowy - średnio różnica pomiędzy programem napisanym w C i asemblerze, to około 15% zajętości pamięci na korzyść asemblera (to jednak zależy od doświadczenia programisty), łatwa lokalizacja błędów i przez to nie tak wiele czasu potrzebnego na uruchomienie programu. Z doświadczenia wiem, że większość - zwłaszcza początkujących programistów - przystępując do wykonania programu liczy tylko czas potrzebny na jego napisanie i zupełnie lekceważy potrzebny na jego uruchomienie. To bardzo duży błąd. Jeden z moich znajomych mawia, że jeśli zapytasz programistę ile potrzebuje czasu i odpowie ci, że na przykład 1 miesiąc, to tworząc harmonogram realizacji całego projektu, należy do czasu podanego przez programistę dodać jeszcze 2-krotną jego ilość, czyli w tym przypadku 3 miesiące. Wiem z praktyki, że tak jest.

Celem tego artykułu nie jest nauczenie programowania w języku C. Jest na ten temat mnóstwo książek, między innymi - już klasyka informatyki - pary autorów Brian Kernigham i Dennis Ritchie „Język ANSI C", traktująca o standardach, które muszą być przestrzegane w każdym z dialektów języka C, bez względu na jego przeznaczenie. To jest opis standardu - z tej książki można nauczyć się podstaw. Do standardu, każdy z autorów kompilatora języka C dołącza rozszerzenia. Moim celem jest opisanie pewnych rozszerzeń języka C, specyficznych dla środowiska, w którym tworzone są aplikacje.

W artykule przedstawię narzędzie, które dobrze znam - pakiet programów firmy Raisonance. Jest to zintegrowane środowisko do tworzenia programów dla mikroprocesorów z rodziny 8051, a również XA firmy Philips i ST62 firmy ST Microelectronics, najpopularniejsza w naszym kraju rodzina mikrokontrolerów.

Zaczynamy - określenie wielkości pamięci

Zaimplementowany w pakiecie Raisonance język RC-51 oferuje 5 różnych modeli kompilacji dla 3 wielkości pamięci programu. Zanim przystąpię do omówienia szczegółów, chciałbym krótko wyjaśnić skąd wzięła się idea różnych modeli kompilacji i modeli wielkości pamięci.

Rolą każdego kompilatora języka programowania wysokiego poziomu (a do takich należy język C) jest przetłumaczenie kodu programu z postaci zrozumiałej dla człowieka, do postaci zrozumiałej dla mikroprocesora, czyli - w dużym uproszczeniu - z języka C do języka asembler. Jeśli prześledzimy uważnie listę rozkazów asemblera dla 8051, zauważymy, że na przykład rozkazy skoków mogą zajmować w pamięci 2 lub 3 bajty (*AJMP* i *JMP*). Podobnie z rozkazami wywołania podprogramów *CALL* i *ACALL*. Rozkazy 2-bajtowe występują wówczas, gdy adres docelowy znajduje się w obrębie tego samego 2kB bloku pamięci programu, a 3-bajtowe gdy w obrębie 64kB. Przedstawiłem to bardzo skrótowo, jednak nasuwają się następujące wnioski. Jeśli pamięć jest mała, to rozkazy skoków mogą być krótsze i możemy w ten sposób zmniejszyć wielkość programu wynikowego! I to jest właśnie jedna z przesłanek do utworzenia różnych modeli kompilacji i różnych modeli pamięci. Dzięki nim kompilator „orientuje się", jaki kod tworzyć i w jaki sposób go optymalizować.

Należy tutaj wspomnieć, że kompilator firmy Raisonance obsługuje również programy o łączonych modelach pamięci. Można więc napisać cały program korzystając z modelu SMALL, a tylko niektóre procedury przy zastosowaniu modelu LARGE i zmiennych umieszczonych w pamięci zewnętrznej. Oczywiście można tak postępować przy spełnieniu pewnych warunków. Jednak - szczególnie jeśli stawiasz pierwsze kroki w języku C - proponuję nie łączyć różnych modeli pamięci.

Korzystanie z pamięci, czyli zmienne i stałe oraz sposoby ich umieszczania w przestrzeni adresowej mikroprocesora

Dla programisty bardzo ważne jest aby zmienne znajdowały się w ściśle określonym miejscu pamięci, pod znanym adresem. RC-51 wyposażono w instrukcje umożliwiające właśnie takie swobodne umieszcza-

Tab. 1. Modele pamięci

Model	Domyślny typ pamięci danych	Wymagana zewn. pamięć danych	Lokalizacja stosu	Uwagi
TINY	Wewnętrzna, do 256 bajtów	* * *	*idata*	Zwłaszcza dla mikroprocesorów serii 8x75x firmy Philips
SMALL	Wewnętrzna, do 256 bajtów	Nie	*idata* lub *pdata*	Możliwe umieszczanie zmiennych w pamięci zewnętrznej
COMPACT	Zewnętrzna, do 64kB	Tak	*idata* lub *pdata*	Taki sam jak LARGE, jednak kod wynikowy jest mniejszy
LARGE	Zewnętrzna, do 64kB	Tak	*idata* lub *pdata*	Doskonały dla dużych aplikacji
HUGE	Zewnętrzna, do 64kB	Tak	*pdata*	Adres powrotu i parametry są zapamiętywane w pamięci zew.

nie zmiennych i stałych. Niestety, nie ma żadnego obowiązującego standardu - prawdopodobnie pochodzące od innych producentów kompilatory C będą wymagały zupełnie innej składni.

Pamięć mikrokontrolera 8051 została podzielona na następujące obszary:
- DATA - bezpośrednio adresowany obszar wewnętrznej pamięci RAM (128 bajtów),
- IDATA - pośrednio adresowany obszar wewnętrznej pamięci RAM mikrokontrolera (128 lub 256 bajtów),
- BDATA - obszar pamięci RAM, w którym możliwe jest adresowanie pojedynczych bitów
- SFR - obszar rejestru funkcji specjalnych w pamięci RAM (*special function registers*),
- SBIT - obszar pojedynczego bitu w obrębie rejestru funkcji specjalnych SFR, na przykład *sbit* OV = PSW^2,
- PDATA - tak samo, jak XDATA z tym, że adres dostępu do pamięci zewnętrznej jest 8-bitowy,
- XDATA - zewnętrzna pamięć danych, przy czym adres dostępu 16-bitowy (64kB),
- CODE - pamięć programu; może to być wewnętrzna, albo zewnętrzna pamięć ROM mikrokontrolera.

Wyżej wymienione słowa kluczowe języka RC-51 noszą nazwę kwalifikatorów obszaru pamięci (*space qualifier*). Innym słowem kluczowym, ściśle z nimi związanym, jest AT.

Spróbujmy teraz powiązać zdobyte już informacje w całość. Spróbujmy dopasować zmienne (także stałe) i modele kompilacji do źródła naszego programu. Jako pierwszy niech posłuży nam przykład programu napisanego dla mikrokontrolera jednoukładowego, który wykorzystuje tylko i wyłącznie swoje zasoby wewnętrzne - nie ma w przestrzeni adresowej dołączonych żadnych urządzeń zewnętrznych.

```
#pragma TINY (lub SMALL)
at 0x20 data char ZMIENNA1;
```

Przede wszystkim określamy, że nasz program zmieści się w pamięci wewnętrznej mikroprocesora AT89C2051, który posiada 2kB pamięci programu. Informuje o tym model pamięci zdefiniowany za pomocą polecenia *#pragma TINY*. Jako druga następuje dyrektywa dla kompilatora, aby zmienną o nazwie *ZMIENNA1* umieścił pod adresem 20H w pamięci danych. Podobnie postępujemy w przypadku pamięci programu:

```
at 0x1000 code char tablica[5]=
{'H','E','L','L','O'};
```

Powyższa instrukcja umieszcza w pamięci programu *CODE* pod adresem 1000H napis HELLO. Oto jeszcze inne przykłady deklaracji zmiennych:

```
char code patterns[10] = {
0x09,0xAF, 0x1A, 0x8A,0xAC,0xC8,
0x48,0x8F,0x08,0x88 };
char code digits[6] = {
0xFE,0xFD,0xFB,0xF7,0xEF,0xDF };
unsigned int data display[6] = {
0xFF,0xFF,0xFF,0x00,0x9,0x09 };
```

Rozpatrzmy inny przypadek, również bardzo często spotykany w praktyce: pamięć programu, stos oraz zmienne wewnątrz mikroprocesora, a na zewnątrz w przestrzeni adresowej (adres 8-bitowy) podłączone są pewne urządzenia zewnętrzne, takie jak: wyświetlacz, klawiatura, port danych urządzenia pomiarowego. Oto przykłady deklaracji:

```
#pragma SMALL;
/* adresy rejestrów wyświetlacza */
at 0x00 pdata char LcdCtrlRegister,
LcdVideoRam, LcdReadCtrl, LcdRead;

/* adresy poszczególnych kolumn
klawiatury */
at 0x34 pdata char Column1;
at 0x2C pdata char Column2;
at 0x1C pdata char Column3;

/* zmienne robocze */
data Nastawy Bufor;
char PRG;
unsigned int Pozycja;
int Zero = 45;
```

Jest to często spotykany przypadek, gdy nie jest istotne gdzie w pamięci danych znajdą się zmienne wykorzystywane w czasie pracy programu oraz stos, ale ważne jest pod jakimi adresami znajdują się podłączone do mikrokontrolera elementy pamięci zewnętrznej. Oczywiście taka deklaracja powoduje, że program będzie pobierał dane *PDATA* używając instrukcji asemblera *MOVX A,@R0*. Gdyby zmienne zadeklarowane zostały jako *XDATA*, wówczas odczytywane byłyby instrukcją *MOVX A,@DPTR*.

Zwróćmy jeszcze uwagę na sposób deklaracji adresów, pod którym znajdują się rejestry wyświetlacza LCD. Taka metoda zapisu adresów oznacza, że *LcdCtrlRegister* znajdzie się pod adresem 0x00, *LcdVideoRam* pod adresem 0x01, *LcdReadCtrl* - 0x02 i tak dalej. Można w ten sposób umieszczać w określonym miejscu pamięci nie tylko zmienne i stałe, ale również funkcje.

```
code at 0x1000 int ZapisDoEEProm
(char x);
code at 0x00FF void UstawBit (void);
```

Bardzo użytecznym kwalifikatorem przestrzeni jest moim zdaniem *sbit*. Dzięki niemu mamy na przykład możliwość powiązania poszczególnych bitów portów wyjściowych mikrokontrolera z ich nazwą symboliczną. Ułatwia to bardzo analizę programu i również jego pisanie.

```
/* funkcje poszczególnych bitów
portu p2 */
sbit Wejscie_1 = P2^4;
sbit Wyjscie_Start = P2^3;
sbit Wyjscie_Kontrola = P2^2;
sbit Wyjscie_Docisk = P2^1;
sbit Wyjscie_Blad = P2^0;
```

Zapis „Wyjscie_Start = 1;“ jest równoważny instrukcji SETB P2.3 a jest bardziej czytelny.

Przerwania

Słowo kluczowe *interrupt* powoduje, że funkcja traktowana jest przez kompilator jako obsługująca przerwa-

Tab. 2. Modele (rozmiary) pamięci programu

Rozmiar pamięci programu	Opis
SMALL	1. Rozmiar programu nie może przekroczyć 2kB 2. Wywołania CALL mają postać ACALL a JMP - AJMP
COMPACT	1. Rozmiar programu nie może przekraczać 64kB 2. Rozmiar funkcji nie może być większy niż 2kB 3. CALL to LCALL, JMP to AJMP
LARGE	1. Rozmiar programu nie może przekraczać 64kB 2. CALL to LCALL, JMP to AJMP

nie. Przy jej definiowaniu wymagana jest znajomość numeru przerwania - kolejności w tablicy wektorów przerwań. Adres funkcji zostanie automatycznie wyliczony. Przeanalizujmy przykładową procedurę obsługi przerwania generowanego przez *Timer 0*.

```
void Przerwanie_Timera0 (void)
interrupt 1
{
 TR0 = 0;   // zatrzymanie timera 0
 TH0 = 0;   // odświeżenie
            // zawartości rejestrów
 TL0 = 0x1F;
 licznik++;// zwiększenie
            // zmiennej licznik o 1
 TR0 = 1;   // ponowne uruchomienie
            // timera 0
}
```

Odpowiedni wektor przerwania obliczany jest jako *3+numer_przerwania x 8*.

Wartości 3 i 8 są przyjmowane jako domyślne dla rodziny 8051. Jeśli zachodzi potrzeba ich zmiany, można to zrobić poprzez modyfikację *INTVECTOR* oraz *INTERVAL* (stałe kompilatora). Program, który jest wykonywany po *RESET* nosi nazwę *main* i do niego zawsze odnosi się wektor przerwania numer 0. Jest to program główny. Kompilator sam dba o to, aby wektor przerwania numer 0 zawsze odnosił się do miejsca w pamięci programu, gdzie umieszczony jest kod wynikowy *main*.

Jaki jest skutek umieszczenia w przykładzie słowa kluczowego *interrupt*? Po pierwsze instrukcja *LJMP Przerwanie_Timera0* jest umieszczana w tablicy wektorów przerwań pod adresem 0BH. Po drugie, przy jej uruchomieniu zapamiętywana jest zawartość rejestrów A, B, PSW, DPH, DPL (na stosie). Po trzecie, na końcu procedury zamiast rozkazu RET, umieszczone zostaje RETI.

Omawiając procedurę obsługi przerwań należy wspomnieć o instrukcji *using*. Służy ona do zmiany banku rejestrów. Często bowiem zdarza się tak, że nasz program główny używa banku rejestrów (R0...R7) do realizacji różnych zadań. Składania polecenia *using* jest następująca:

```
void Przerwanie_Timera0 (void)
interrupt 1 using 1
int Pomnóż_Liczby (char x,y) using 2
```

To, co najbardziej decyduje o elastyczności języka C - wskaźniki

Wskaźnik (*pointer*) to zmienna, która zawiera adres innej zmiennej. Użycie wskaźników prowadzi do bardziej efektywnego kodu niż otrzymywany innymi metodami. Jedną z niepożądanych cech wskaźników jest to, że powodują często tworzenie zupełnie niezrozumiałych programów. Dzieje się tak zwłaszcza wtedy, gdy wskaźniki stosowane są w sposób „nieostrożny". Łatwo jest dla przykładu utworzyć wskaźnik, który będzie wskazywał nie wiadomo co i nie wiadomo jakiego typu. Jednak przy przestrzeganiu pewnych zasad, wskaźniki stają się jednym z najmocniejszych mechanizmów języka C.

Kompilator RC-51 oferuje nam 2 typy wskaźników. Pierwszy, to wskaźnik obszaru pamięci deklarowanego za pomocą wcześniej omawianych kwalifikatorów (takich jak *CODE* czy *DATA*). Drugi umożliwia wskazanie dowolnej danej w jakimkolwiek obszarze adresowania mikrokontrolera. Ważne jest, że jeśli wskaźnik przypisany jest do obiektu zakwalifikowanego do określonego obszaru pamięci, to również zostaje przypisany do tego właśnie obszaru pamięci, w którym umieszczony jest wskazywany obiekt. Drugi typ jest specyficznym typem wskaźnika, rozszerzeniem wprowadzonym przez firmę i nosi nazwę *generic*. Dlaczego - mógłby ktoś zapytać - nie stosować wyłącznie wskaźników typu *generic*? Oczywiście, można to zrobić. Główną jednak przyczyną wprowadzenia takiego podziału jest to, że typowo wskaźnik *generic* zajmuje 3 bajty, natomiast kwalifikowany do pewnego obszaru pamięci mikrokontrolera - 2 bajty. Wskaźnik typu *generic* zawiera dodatkowo 1 bajt przeznaczony na kod wskazywanego obszaru pamięci.

```
data char[10] Tablica =
{0,1,2,3,4,5,6,7,8,9}
char generic *Tablica;
// wskaźnik do obszaru DATA,
// wskazuje elementy
// zmiennej Tablica
char generic *p1;
// wskaźnik do dowolnego obszaru
// danych
pdata char *klawiatura;
// wskaźnik do 8-bitowego adresu
// urządzenia zewnętrznego
xdata char *pamiec_danych;
// wskaźnik do zewnętrznej pamięci
```

W powyższym przykładzie * jest operatorem adresowania pośredniego. Zastosowany do wskaźnika podaje zawartość wskazywanego obiektu. W języku C istnieje jeszcze inna metoda przypisania wskazań do określonej zmiennej. Jednoargumentowy operator & zwraca adres zmiennej (uwaga: można go stosować tylko do obiektów zajmujących pamięć zmiennych oraz tablic).

```
char x = 1, y = 2, z[5];
char *wskaznik;

ip = &x;
//zmienna wskaznik wskazuje na x
y = *wskaznik;
//teraz zmienna y ma wartość 1
*wskaznik = 0;
//teraz zmienna x ma wartość 0
wskaznik = &z[0]
//zmienna wskaznik wskazuje na
//pierwszy element tablicy z
```

W deklaracji wskaźnika powinno być naśladowanie elementu, do którego odnosi się wskazanie.

Zastosowanie wskaźników znacznie upraszcza program. Dobrze znana jest programistom konieczność przekazywania parametrów do procedur. W wielu językach programowania parametry przekazuje się wymieniając długą ich listę. Zajmuje to bardzo dużo miejsca w pamięci RAM. W języku C tę listę parametrów, łącznie z długimi tablicami (buforami danych), można przekazać przez proste wskazanie na adres w pamięci, gdzie ta zmienna się znajduje. Jest to jeden z głównych powodów, dla którego program napisany w języku C może być wykonywany bardzo szybko, przy oszczędnym gospodarowaniu stosem mikrokontrolera.

Wskaźniki i argumenty funkcji

W języku C parametry wywołania funkcji przekazywane są przez wartość. Na skutek tego funkcja (działania wewnątrz tzw. ciała funkcji) nie ma dostępu do argumentów należących do wywołującego ją podprogramu. Jedyny sposób pozwalający na wykonanie działań na zmiennych pochodzących z innego podprogramu, to przekazanie jako argumentów wywołania funkcji, wskaźników do tych zmiennych.

Operator	Przykład użycia	Opis
Operatory arytmetyczne		
+	x + y	Dodawanie
-	x - y	Odejmowanie
*	x * y	Mnożenie
%	10 % 3	Reszta z dzielenia (w przykładzie: wynik = 1)
Operacje arytmetyczne		
+=	x += 10	x = x + 10
-=	x -= 10	x = x - 10
*=	x *= 3	x = x * 3
/=	x /= 23	x = x / 23 (dzielenie)
%=	x %= 2	reszta z dzielenia przez 2
^=	x ^= 2	x = x XOR 2
|=	x |= 0xF0	x = x OR F0H
<<=	x <<= 2	x = x << 2 (przes. 2 razy w lewo)
>>=	x >>= 4	x = x >> 4 (przes. 4 razy prawo)
Operatory logiczne		
>	if (x > 23)	Znak większości
>=	If (x >= 10)	Znak większe-równe
<	if (x < 10)	Znak mniejszości
<=	if (x <= y)	Znak mniejsze-równe
==	if (x == 0)	Porównanie
!=	if (x != 0)	Znak różności
Operacje logiczne		
||	if (x || 0x11 = 0x11)	Suma logiczna (OR)
&&	if (x && 0x80 = 0)	Iloczyn logiczny (AND)
!	!x	Negacja logiczna (NOT)
Operatory bitowe		
|	P1 = P1 | 0xF0	Suma bitowa
&	P1 = P1 & 0xF0	Iloczyn bitowy
^	P1 = P1 ^ 0xAA	Bitowe wyłącznie-lub
<<	A = A << 4	Przesunięcie w lewo 4 razy
>>	A = A >> 2	Przesunięcie w prawo 2 razy
~	~0x77	Dopełnienie jedynkowe

```
void Zamien (int *Tx, int *Ty)
{
 int temp;

 temp = *Tx;
 *Tx = *Ty;
 *Ty = temp;
}

void main(void)
{
 int A = 10, B = 20;
 Zamien(&A, &B);
//zamiana wartości zmiennych A i B
}
```

Do tego potrzebujemy tylko 2 bajty na przekazanie parametrów, bowiem nasze zmienne leżą w obszarze pamięci DATA. A poza tym działania wewnątrz funkcji są wykonywane bezpośrednio na zmiennych programu głównego. Żadnych „pośredników", np. w formie stosu, nie potzreba.

Kilka słów o arytmetyce wskaźników

Zasady arytmetyki wskaźników są bardzo proste. Jeśli wskaźnik T pokazuje początek tablicy elementów typu *int* i sam również jest typu **int*, to wówczas operacja T=T+1; (inaczej T++;) przesunie wskazanie na następny element tablicy. Do adresu wskazania nie zostanie dodana liczba 1 lecz 2, ponieważ zmienna typu *int* ma 2 bajty długości. Wszystkie operacje na wskaźnikach są automatycznie dostosowywane do rozmiaru wskazywanych obiektów.

Do operacji wskaźnikowych należą: przypisanie wskaźników do obiektów tego samego typu, dodawanie i odejmowanie wskaźnika i liczby całkowitej, odejmowanie bądź porównywanie dwóch wskaźników z elementami tej samej tablicy oraz przypisanie wskaźnikowi wartości 0 lub przyrównanie wskaźnika do 0. Wszystkie inne operacje na wskaźnikach są nielegalne. Nie wolno dodawać do siebie dwóch wskaźników (nawet tego samego typu) ani ich mnożyć, dzielić, przesuwać w prawo bądź w lewo, składać z maskami (operacje AND i OR), ani też dodawać do nich liczb typu *float* i *double*. Nie wolno nawet - z wyjątkiem wskaźnika typu **void* - wskaźnika obiektu jednego typu przypisać bez przekształcenia (rzutowania) do obiektów innego typu.

Wskaźniki w języku C to temat bardzo obszerny. Zainteresowanych poszerzeniem swojej wiedzy oraz praktycznymi zastosowaniami wskaźników, zapraszam do korespondencji ze mną oraz lektury materiałów źródłowych na ten temat.

To, co przysparza mi zawsze najwięcej kłopotów w języku C - operatory

Kolejnymi elementami języka C są operatory podstawowych operacji: dodawania, odejmowania, sumy i iloczynu logicznego i tak dalej. Osobiście zawsze miałem kłopoty z ich zapamiętaniem. Dlatego też, zwłaszcza dla początkujących, zdecydowałem się je zestawić w tabeli. Skopiujcie ją sobie, bo jeszcze nie raz do niej wrócicie. Zwłaszcza analizując programy pisane przez kogoś innego.

Porty I/O, działania na portach

Korzystając z portów I/O mikrokontrolera w programach napisanych w języku C należy kierować się tymi samymi zasadami, co w języku asembler. To znaczy porty, które wymagają ustawienia (poziomy wysokie) przed odczytem z nich danych, muszą być w ten stan ustawione. Nic nie stanie się bez naszego udziału, niczego kompilator nie zrobi za nas. Gdy o tym pamiętamy, to korzystanie z portów jest bardzo łatwe. Można testować wartości bitów, przypisywać zmienne i stałe, wykonywać różne inne operacje. Podobnie jak w asemblerze, jeśli operacja wymaga zmian stanu portu (przesunięć bitów i podobnych operacji), lepiej jest utworzyć zmienną będącą kopią stanu portu i na niej wykonywać działania. Potem wystarczy tylko przypisać do danego portu wartość zmiennej - unikniemy w ten sposób błędów związanych z zakłóceniami jakie mogą wystąpić na wyprowadzeniach portu.

- odczyt stanu portu:

```
1.  zmienna = P1
    // (0..3 - w zależności od
    // mikrokontrolera)
2.  P1 = P1 | 0xFF;
    // gdy port pracuje jako
    // wejściowy, dobrze jest jego
    // bity ustawić na "1":
    // zmienna = P1;
```

- zapis do portu:

```
1.  P1 = 0xAA;
    // (numer portu zależny od
    // aplikacji),
2.  P1 = P1 | 0x01;
    // ustawienie bitu P1.0
3.  P1 = P1 & 0xFE;
    // wyzerowanie bitu P1.0
4.  P1^0 = 1;
    // odpowiednik rozkazu SETB P1.0
5.  P1^0 = 0;
    // odpowiednik rozkazu CLR P1.0
```

- testowanie stanu bitu:

```
    P1 = P1 | 0x01;
    // ustawienie P1.0 na "1"
    if (P1^0 = 1....
    // jeden ze sposobów testowania
    //stany P1.0
```

Po tym nieco przydługim wprowadzeniu najwyższy czas aby zaprezentować przykłady zastosowań. Będzie można wówczas najlepiej zobaczyć, w jaki sposób wykonuje się działania na bitach, zmiennych, portach itp. Pokażemy to w kolejnej części kursu.

Jacek Bogusz, AVT
jacek.bogusz@ep.com.pl

Dodatkowe informacje

Ewaluacyjna wersja pakietu firmy Raisonance znajduje się na CD-EP6/2002B.