Bascom czy C?

część 3

W trzeciej (ostatniej) części artykułu kontynuujemy porównanie dwóch bardzo popularnych języków programowania mikrokontrolerów: Bascoma i C/C++.
Nie odpowiadamy na pytanie: który z nich jest lepszy? Ocenę i wybór pozostawiamy Czytelnikom.

Pętle

Basic i C oferują nam kilka typów pętli. Pierwsza z nich to `do-loop` zaimplementowana w Basicu:

```
Do
instrukcje
Loop Until warunek
```

Jej działanie jest takie: wykonaj `instrukcje` i sprawdź, czy `warunek` jest prawdziwy. Jeśli nie jest, pętla jest powtarzana. Można także wyjść z pętli, używając instrukcji `Exit Do`:

```
Do
instrukcje
If warunek_2 Then
    Exit Do
Loop Until warunek_1
```

Można nawet pominąć `until warunek`, wtedy pętla będzie wykonywana bez końca (choć można z niej wyjść za pomocą `Exit Do`):

```
Do
rob_cos_bez_konca
Loop
```

Odpowiednikiem tej pętli w C jest pętla `do-while` mająca postać:

```
do
    instrukcja;
while(warunek);
```

Pętla ta jest wykonywana, dopóki `warunek` jest prawdziwy. Jeżeli w jej wnętrzu jest więcej niż jedna instrukcja, należy je objąć nawiasami klamrowymi:

```
do
{
    instrukcja1;
    instrukcja2;
}
    while(warunek);
```

Z pętli można także wyjść używając instrukcji `break`:

```
do
{
    ...
    if (warunek_2)
        break;
    ...
}
while(warunek_1);
```

Dodatkowo w języku C w pętlach dostępne jest polecenie `continue`, które powoduje bezwarunkowy skok do instrukcji sprawdzającej warunek:

```
do
{
    if (warunek_2)
        continue;

    instrukcja;
}
while(warunek_1);
```

W powyższym przykładzie, jeżeli w danym przejściu pętli warunek_2 jest prawdziwy, to zostanie wykonana instrukcja `continue`, co spowoduje pominięcie (w tym jednym konkretnym przejściu pętli) wszelkich następnych instrukcji (w tym przypadku, po prostu nie zostanie wykonana `instrukcja`). `Break` i `continue` są dostępne we wszystkich trzech rodzajach pętli w C. Jeżeli chcemy mieć pętlę wykonywaną bez końca, możemy jako warunek podać wyrażenie, które jest zawsze prawdziwe, np.:

```
do
instrukcja;
while(1);
```

W obu językach warunek jest konstruowany w oparciu o te same reguły, co w instrukcji `if`. W Basicu – w przeciwieństwie do języka C – nie można go tworzyć w oparciu o bardziej złożone formuły, np. zawierające operacje arytmetyczne. Dostępne są jedynie operacje logiczne typu `and`, `or`, `not`. Zarówno `do-loop`, jak i `do-while` są wykonywane co najmniej raz. Wynika to z faktu, że sprawdzenie warunku odbywa się po wykonaniu instrukcji zawartych w pętli. W obu językach występuje także drugi typ pętli, bardzo podobny, a różniący się właśnie tym, że pętla jest wykonywana dopiero po sprawdzeniu warunku. W Basicu jest to pętla `while-wend`, z której można wyjść, korzystając z instrukcji `Exit While`:

```
While warunek
instrukcje
Wend
```

W C jest to pętla `while`:

```
while(warunek)
    instrukcja;
```

lub gdy wykonujemy więcej niż jedną instrukcję:

```
while(warunek)
{
    instrukcja1;
    instrukcja2;
}
```

Trzecim rodzajem pętli jest pętla `for`. W Basicu wygląda ona tak:

```
For i = 1 To 10
instrukcja1
instrukcja2
Next i
```

Wszystkie instrukcje znajdujące między `For` a `Next` zostaną wykonane w tym wypadku 10 razy. Po każdym przejściu i będzie zwiększane o 1, dopóki nie osiągnie wartości 10. Można to zmienić, stosując dodatkowy parametr `Step`:

```
For i = 1 To 10 Step 2
instrukcje
Next i
```

Tym razem i będzie zwiększane za każdym razem o 2. Można także zupełnie zmienić kierunek zliczania:

```
For i = 10 To 1 Step -2
instrukcje
Next i
```

W tym przypadku `i` będzie miało na początku wartość 10 i będzie zmniejszane o 2, dopóki nie osiągnie wartości równej 1. W pętli `for` wystarczy podać zmienną i dwie wartości. Stosowanie `Step` jest opcjonalne. Można z niej wyjść dzięki instrukcji `Exit For`.

W języku C pętla ta skonstruowana jest zupełnie inaczej. Składa się ona z trzech części. Każda z nich to jedna instrukcja. Może być nią przypisanie lub wywołanie funkcji – w zasadzie wszystko jest dozwolone. Pierwsza instrukcja wykonywana jest raz – przed pierwszym wejściem do pętli. Następna wykonywana jest przed każdą pętlą. Może to być test sprawdzający, wywołanie funkcji itp. Nie ma tu ograniczeń. Z reguły jest to test, który sprawdza, czy wykonać kolejną pętlę, czy nie. Ostatnia instrukcja jest wykonywana po każdym obiegu pętli. Zazwyczaj jej zadanie to zmiana wartości zmiennej, której powierzono rolę licznika. Prosta pętla może mieć postać:

```
for (i=0; i<10; i++)
    instrukcja;
```

Zadaniem pierwszej część pętli: `i=0` jest przypisanie wartości 0 zmiennej `i`. Druga część: `i<10` stanowi test. Jeżeli jego wynik jest prawdziwy, pętla zostanie wykonana. Ostatnia część: `i++` to zwiększenie o 1 zmiennej `i` po zakończeniu każdego obiegu pętli. Jeżeli chcemy, by licznikiem była zmienna już posiadająca jakąś konkretną wartość początkową, możemy pominąć pierwszą część pętli `for`, zastępując ją średnikiem:

```
for (; i<16; i++)
    instrukcja;
```

Jeżeli chcemy liczyć w dół, to należy sprawdzać, czy licznik jest większy od ustalonej dolnej granicy, a po każdej pętli zmniejszać go:

```
for (i=20; i>10; i--)
    instrukcja;
```

Mamy także wpływ na to, o ile licznik zostanie zmieniony po przejściu przez każdą pętlę:

```
for (i=1; i<10; i*=4)
    instrukcja;
```

Tym razem zwiększamy go za każdym razem czterokrotnie. Poszczególne instrukcje mogą być np. wywołaniami do funkcji, może ich także w ogóle nie być. Pętla `for` może z powodzeniem zastępować pętle `do-while` i `while`. Na przykład jeżeli chcemy wykonywać jakąś czynność, dopóki nie nadejdzie konkretna komenda portem szeregowym, który trzeba najpierw zainicjalizować, możemy zastosować właśnie pętlę `for`. Załóżmy, że funkcja `void InitUART()` zainicjuje ten port, a funkcja `unsigned char GotMsg ()` zwróci 1,

gdy nadejdzie ta konkretna komenda. Zapiszemy więc:

```
for (InitUART(); GotMsg();)
    instrukcja;
```

Trzeciej części pętli nie ma – nie jest ona potrzebna. Jeżeli mikrokontroler ma po prostu krążyć w pętli, dopóki nie odbierze tego znaku, można to zapisać tak:

```
for (InitUART(); GotMsg(););
```

Ostatni średnik po instrukcji `for` oznacza „nie rób nic”. Ta pseudoinstrukcja „;” nie będzie miała żadnego odzwierciedlenia w skompilowanym kodzie. Jest on stosowany właśnie w takich specyficznych przypadkach jak ten. Równie dobrze można zastosować poniższy zapis:

```
for (InitUART(); GotMsg();)
{
    /* między nawiasami klamrowymi nie ma
       żadnej instrukcji */
}
```

Typowa dla języka C jest również pętla `for` postaci:

```
for (;;)
   instrukcja;
```

Taki zapis jest dokładnym odpowiednikiem:

```
while(1)
   instrukcja;
```

lub w Basicu:

```
Do
   instrukcja
Loop
```

Zasięg zmiennych

Między Basikiem i C występuje bardzo znacząca różnica dotycząca deklaracji zmiennych i ich „widoczności” z różnych fragmentów programu. W Basicu występują zmienne globalne i lokalne. Globalne są widoczne z każdego miejsca w programie. Lokalne – tylko z wnętrza funkcji i procedur, w których zostały zadeklarowane. Przykład:

```
Declare Sub Procedura
Dim A As Byte
    'tu widać zmienną A
For A = 1 To 5
    Dim B As Byte
    'tu widać zmienne A i B
Next A
    'tu widać zmienne A i B
End
Sub Procedura
   Local C As Byte
   'tu widać lokalną zmienną C, ale także
   'globalne A i B
End Sub
```

W języku C są także zmienne globalne i lokalne, lecz znaczenie tych określeń jest trochę inne. To, czy dana zmienna jest globalna czy lokalna, zależy od miejsca w programie, z którego na nią patrzymy. Dlatego nie są stosowane w praktyce pojęcia „globalny” i „lokalny”. Prześledźmy kilka przykładów zmiennych postrzeganych jako lokalne:

```
void funkcja()
{
  int a; /*zmienna widziana tylko w tej funkcji*/
}
```

albo np. w pętli `for`:

```
for (i=0; i<10;i++)
{
    int a;
    /* zmienna a jest widziana tylko tu */
}
/* tu nie widać zmiennej a */
```

Zmienne zadeklarowane poza wszystkimi funkcjami widziane są z wnętrza wszystkich funkcji (są to zmienne globalne):

```
int a; /* to jest zmienna globalna */

void funkcja()
{
    int b;
    /* tu widać zmienne a i b */

    if (1)
    {
       int c;
       /* tu widać zmienne a, b i c */
    }
}

int main()
{
    int b;
/* tu widać zmienne a i b, lecz zmienna b
to nie ta, która jest zadeklarowana w funkcji
„funkcja” */
}
```

Na tych przykładach można zauważyć, że klamerki ograniczają „widoczność” zmiennych, tworząc strukturę podobną do drzewka. Czynnikami niejako wymuszającymi korzystanie z klamerek są odpowiadające im konstrukcje, np. funkcje i instrukcja `if`. Klamerki można jednak stosować także bez tych instrukcji:

```
int main()
{
   int a;
   /* tu widoczna jest zmienna a */
   {
       /* tu widoczna jest zmienna a */
       int b;
       /* tu widoczne są zmienne a i b */
       {
          /* tu widoczne są zmienne a i b */
          int c;
          /* tu widoczne są zmienne a, b i c */
       }
       /* tu widoczne są zmienne a i b */
   }
   /* tu widoczna jest zmienna a */
}
```

Dodatkowo, na każdym kolejnym poziomie zagnieżdżenia klamerek możemy deklarować zmienne o nazwach takich samych jak te, które już zostały zadeklarowane wcześniej (umownie przypiszemy im dla rozróżnienia numery 1 i 2):

```
int main()
{
   int a; /* niech ta zmienna ma nr 1 */

   a = 12; /* to przypisanie odnosi się do
              zmiennej a nr 1 */

   if (jakis_warunek)
   {
       a = 13; /* to przypisanie odnosi się
                  do zmiennej a nr 1 */
       int a; /* niech ta zmienna ma nr 2 */
       a = 100; /* to przypisanie odnosi się
                   do zmiennej a nr 2 */
       /* zmienna a nr 1 cały czas ma wartość
          13, lecz z tego miejsca nie możemy
          się do niej odnieść */
   }
   /* tutaj nie widać zmiennej a nr 2 */
   a = 33; /* teraz zmienna a nr 1 ma wartość
              33 */
}
```

Struktury i unie – specyficzne typy danych w języku C

Bascom, jako język, został stworzony specjalnie do programowania na dwa konkretne typy mikrokontrolerów, dlatego jego integralną częścią są wszystkie instrukcje odwołujące się bezpośrednio do sprzętu – upraszczają one bardzo dostęp do timerów, portów szeregowych, a nawet software'owo emulują interfejsy takie jak I²C czy 1-Wire, umożliwiają łatwą obsługę alfanumerycznych i graficznych wyświetlaczy LCD. Ostatnio nawet zaimplementowano wsparcie dla protokołów sieciowych.

Język C nie jest przyporządkowany żadnej konkretnej platformie. Jednak dysponuje on bardzo obszernym zbiorem funkcji. Znajdują się one w tzw. bibliotece standardowej, która jest dołączana do każdego kompilatora. Oczywiście nie jest ona zawsze taka sama. Nie będę tu podawał ani porównywał żadnych z tych funkcji. Celem tego artykułu jest porównanie samych języków. Trudno oczekiwać, by w bibliotece dołączonej do kompilatora przeznaczonego dla środowiska Windows znajdowały się wywołania obsługujące przerwania w mikrokontrolerach PIC albo żeby w jej wersji dla AVR można było znaleźć implementację procesów i wątków dla systemu Linux. Jest jednak zbiór funkcji, które powinny się znajdować w każdej z tych implementacji. Niektóre z nich nie zawsze są dostępne dla mikrokontrolerów. Na przykład w niektórych kompilatorach nie mamy do dyspozycji funkcji `malloc()` lub `free()`, które są odpowiedzialne za dynamiczne alokowanie i zwalnianie bloków pamięci już podczas działania programu. Stanowią one podstawę działania wszelkich programów dla „większych braci” mikrokontrolerów – x86 itd. Te różnice doprowadziły do tego, że dostępne kompilatory wprowadzają swoje „dodatki” do standardowej specyfikacji C, np. wymieniony wcześniej typ danych `bit`. Większość kompilatorów C zaopatrzono w biblioteki implementujące te elementy języka, które stanowią integralną część Basica. Są to funkcje matematyczne (np. sin), funkcje operujące na ciągach znaków, zamieniające zmienne liczbowe na ciągi znaków i *vice versa*, umożliwiające dostęp do pamięci EEPROM, obsługujące wyświetlacze LCD i wiele innych elementów typowych dla mikrokontrolerów. C to język, który zawsze jest blisko sprzętu (jak określają to użytkownicy), choć można w nim pisać programy na tak skrajnie różne maszyny jak np. '51 i x86. Jego konstrukcje są tak przemyślane, że zachowują dużą elastyczność, a jednocześnie w pewnym sensie wymuszają taki sposób pisania programów, że kompilator może bardzo dobrze zoptymalizować kod wynikowy. W przypadku mikrokontrolerów, na ogół wymaga on od programisty pewnej wiedzy na temat sprzętu, na którym ten kod ma działać. Basic upraszcza wiele zagadnień do niezbędnego minimum. Umożliwia wręcz pisanie programów, traktując mikrokontroler jako „czarną skrzynkę”. Stanowi to ogromne ułatwienie – szczególnie dla początkujących programistów. Wielu z nich odstrasza fakt, że muszą uczyć się znaczeń poszczególnych bitów

rejestrów specjalnych i wielu innych „okropieństw”, zamiast polutować kilka kabelków, włączyć zasilanie i zobaczyć na wyświetlaczu upragnione „Hello, world!”. Wydaje mi się jednak, że przy takim podejściu można szybko dotrzeć do kresu możliwości Basica. Od wielu lat programuję w C/C++ (choć w 95% na x86). Przyzwyczaiłem się do tego, że C i C++ umożliwiają zrobienie praktycznie wszystkiego, na co pozwala sam procesor lub mikrokontroler. Na początku swojego kontaktu z Basikiem byłem szczerze zachwycony jego prostotą i funkcjonalnością. Jednak dopiero później zaczęło mi brakować mechanizmów typowych dla C i C++. Chociażby wskaźniki – ktoś może zapytać „A po co mi to?”. Nie będę tu starał się nikogo przekonać, że wskaźniki są użyteczne – dopiero kiedy zacznie się je stosować w praktyce, okazuje się, co tak na prawdę można dzięki nim zrobić. W języku C nie trzeba niczego obchodzić naokoło, gdy chce się zrobić coś bardziej nietypowego, coś, czego nie przewidziano podczas projektowania samego języka. Jest tu szereg elementów, których w ogóle nie ma w Basicu. Przykładem mogą być np. takie typy danych, jak struktury i unie (o klasach nawet nie wspomnę, choć są podstawą przy programowaniu np. na x86). Struktury to typy danych, które tworzy się samemu, by opisać np. zdarzenie lub obiekt charakteryzujący się wieloma właściwościami. Posłużę się prostym przykładem. Do opisania na przykład napięcia występującego w pewnym układzie elektrycznym wystarczy jedna zmienna. Często jednak oprócz napięcia musimy podać również prąd płynący w danej gałęzi tego obwodu. W takim przypadku ujawnia się cała moc programowania obiektowego. Wymyślono je po to, by łatwiej było zorganizować zarówno kod, jak i pamięć. Znowu posłużę się przykładem. Gdy chcemy zrobić rejestrator, który ma co 10 sekund przez 1 minutę mierzyć napięcie w danym miejscu, należy zrobić tablicę, która przechowa wyniki tych pomiarów. W Basicu zrobimy to tak:

```
Dim pomiary(6) As Single
```

w C natomiast:

```
float pomiary[6];
```

Gdy chcemy rejestrować napięcie i prąd, należy użyć dwóch osobnych tablic – w Basicu wyglądałoby to tak:

```
Dim napiecie(6) As Single
Dim prad(6) As Single
```

W C możemy zrobić tak samo, lecz dużo wygodniejsze będzie zastosowanie struktury – nazwijmy ją `Spomiar`. Będzie ona jednocześnie nowym typem zmiennych:

```
struct SPomiar
{
    float napiecie, prad;
};
```

Struktura ta przechowuje w sobie dwie zmienne typu `float`: `napiecie` i `prad`. Teraz robimy tablicę:

```
struct SPomiar pomiary[6];
```

Aby odwołać się do piątego elementu tej tablicy i przypisać zmiennym `napiecie` i `prad`, konkretne wartości piszemy tak:

```
pomiary[4].napiecie = 22.3;
pomiary[4].prad = 0.003;
```

Taka organizacja danych pozwala na pogrupowanie zmiennych w tzw. obiekty, co ułatwia na przykład przekazanie wyników pomiarów do funkcji, która wypisze je na wyświetlaczu:

```
wypisz_wyniki(pomiary);
```

Funkcja taka musiałaby zostać zadeklarowana jak poniżej (dokładny opis funkcji `printf` wykracza poza ramy tego artykułu):

```
void wypisz_wyniki(SPomiar *wyniki)
{
/* tu ze zmiennej wyniki można korzystać jak
z normalnej tablicy */
    char i=0;
    for (;i<6;i++)
       printf(„Pomiar %d:\nU = %f\nI = %f\n”,
            wyniki[i].napiecie, wyniki[i].prad);
}
```

A jeżeli chcielibyśmy zamiast ze struktur korzystać z dwóch osobnych tablic:

```
wypisz_wyniki(float *napiecia, float *prady)
{
    char i=0;
    for (;i<6;i++)
       printf(„Pomiar %d:\nU = %f\nI = %f\n”,
            napiecia[i], prady[i]);
}
```

Łatwo jednak zauważyć, że jeżeli chcemy dodać kolejny parametr pomiarów, to trzeba zmieniać deklaracje wszystkich funkcji takich jak `wypisz_wyniki`, a ponadto ich późniejsze wywołania zaczną się rozwlekać. Zastosowanie struktur umożliwia także skrócenie czasu wywołania samej funkcji – w powyższym przykładzie przekazywany jest tylko jeden parametr. Struktury mogą być zagnieżdżane: jedna struktura ma w sobie (jako zmienną) inne struktury, a te z kolei mogą mieć następne.

Niezwykle użytecznym elementem języka C, niemającym swojego odpowiednika w Basicu, są unie. W strukturze wszystkie elementy mają swoje osobne miejsce w pamięci – są oddzielone od siebie, tak jak wszystkie zmienne. W unii natomiast wszystkie elementy pokrywają się. Przyjrzyjmy się poniższemu zapisowi:

```
union Unia
{
    unsigned int A;
    struct{
          unsigned char b, c;
    };
};
```

Teraz 2 bajty zajmowane przez zmienną `A` będą dzieliły tę samą pamięć, co struktura zawierająca zmienne `b` i `c` – będą one zajmowały młodszy i starszy bajt zmiennej `A`. Unie mogą okazać się przydatne w momencie, kiedy kompilator nie obsługuje C++, a jedynie C. Zbudujmy drugi rejestrator. Ma on przechwytywać dwa rodzaje zdarzeń. Gdy nastąpi jedno, zapamiętujemy poziom napięcia. Gdy drugie – stan 32 wejść logicznych. Nie wiadomo jednak, kiedy i w jakiej kolejności one nastąpią. Rejestrujemy także czas – dla stanów logicznych w milisekundach, dla napięcia – w sekundach (a konkretnie w ułamkach sekund). Zależy nam także na maksymalnym wykorzystaniu dostępnej pamięci RAM i naszej wygodzie. Deklarujemy strukturę:

```
struct SWydarzenie
{
    char Typ;
    union
    {
        struct
        {
            unsigned long uCzas;
            unsigned long uStany;
        };
        struct
        {
            float fCzas;
            float fNapiecie;
        };
    };
};
```

Dzięki unii struktura ze zmiennymi `uCzas` i `uStany` zajmuje te same 8 bajtów pamięci co struktura ze zmiennymi `fCzas` i `fNapiecie`. Powiedzmy, że `Typ` będzie się równał 0 dla stanów logicznych i 1 dla napięcia. Robimy tablicę o wielkości np. 10 zdarzeń:

```
struct SWydarzenie wydarzenia[10];
```

Parametry wydarzeń zapisujemy tak (cyfrowe):

```
wydarzenia[2].Typ = 0;
wydarzenia[2].uCzas = 3245;
wydarzenia[2].uStany = PORTA;
```

i analogowe:

```
wydarzenia[8].Typ = 1;
wydarzenia[8].fCzas = 0.33;
wydarzenia[8].fNapiecie = 2.422;
```

Tablica wydarzenia może zajmować całą dostępną pamięć RAM. Zawsze mamy pewność, że niezależnie od tego, ile których zdarzeń nastąpi, to pamięć ta się nie zmarnuje. W Basicu nie da się tego tak prosto zrobić – jedyne rozwiązanie to samodzielne opracowanie mechanizmu umożliwiającego przechowanie w pamięci parametrów dwóch różnych typów wydarzeń, a jedyne wsparcie ze strony samego języka to funkcje `Inp`, `Out` i Varptr. Zmienna `Typ` konieczna jest do identyfikacji – inaczej nie wiedzielibyśmy, jaki typ wydarzenia przechowuje dany element tablicy. Do ich rozróżnienia wybraliśmy liczby 0 i 1. Jest to trochę nieczytelne. Dlatego można zrobić tak:

```
#define WydCyfrowe 0
#define WydAnalogowe 1
```

Teraz gdziekolwiek w kodzie pojawi się odniesienie do jednej z tych nazw, zostanie ona zamieniona na odpowiadającą jej liczbę:

```
wydarzenia[2].Typ = WydCyfrowe;
```

W czasie kompilacji wyrażenie `WydCyfrowe` zostanie zamienione na liczbę 0 (nie jest to jednak stała, choć mogłoby się tak wydawać). Gdy mamy więcej takich deklaracji albo gdy nie wiadomo, ile ich będzie, wygodniejsze jest zadeklarowanie swojego typu zmiennych (w tym przypadku o nazwie `TypWydarzenia`):

```
typedef enum{WydarzenieCyfrowe, WydarzenieAnalogowe}
      TypWydarzenia;
```

Teraz zamiast:

```
char Typ;
```

napiszemy:

```
TypWydarzenia Typ;
```

Konkretną wartość przypiszemy w naturalny sposób:

```
Typ = WydarzenieAnalogowe;
```

W języku C deklarowanie własnych typów zmiennych jest często spotykaną praktyką. Na przykład, jeżeli ktoś jest przyzwyczajony do określania 2-bajtowej

zmiennej bez znaku jako „Word”, może napisać tak:

```
typedef unsigned int Word;
```

Teraz może korzystać ze słowa `Word` zamiast `unsigned int`:

```
Word zmienna;
```

Podsumowanie

Bascom tworzy uniwersalne narzędzie do pracy z mikrokontrolerami ‘51 i AVR. Prostota jego instrukcji i wbudowana obsługa wielu urządzeń sprawia, że pierwsze urządzenie z wykorzystaniem mikrokontrolera nie powstaje po wielu nocach spędzonych na czytaniu opisów instrukcji asemblerowych itp. Do niedawna jednym z głównych zadań pierwszego urządzenia z mikrokontrolerem było najczęściej miganie diodą LED. Obecnie coraz częściej urządzenie to wypisuje na wyświetlaczu LCD tekst „Hello, World!”. Na dodatek do napisania takiego programu nie jest wymagana (prawie) żadna wiedza na temat działania samego mikrokontrolera i wyświetlacza LCD. Bardziej zaawansowani programiści też mogą odnieść wiele korzyści z pracy z Bascomem – chociażby ze względu na fakt obsługi wielu urządzeń i protokołów. Ponadto może on znacznie skrócić czas od pomysłu do realizacji. Jego prostota jest dużą zaletą, lecz może przerodzić się w równie wielką wadę. Jeżeli chcemy podłączyć do mikrokontrolera urządzenie, np. układ scalony lub cokolwiek innego, czego Bascom nie obsługuje – trzeba samemu zaimplementować wszystkie funkcje itp. Zrobienie tego bez pewnej znajomości zarówno urządzenia, jak i protokołu, którym się komunikuje oraz samego mikrokontrolera jest bardzo trudne (a czasami nawet niemożliwe). Wtedy właśnie ktoś, kto przyzwyczaił się, że Bascom zrobi za niego wszystko, stanie przed trudną do pokonania przeszkodą. Język C stworzono w zupełnie innych warunkach i opierając się na zgoła innych założeniach. Były to czasy, gdy dopiero powstawały komputery znane dziś jako PC, a każdy bajt pamięci był na wagę złota – stąd skrótowe formy poleceń i++ zamiast Incr i, i+=10 zamiast i=i+10 itd. Programy pisane w języku wysokiego poziomu musiały działać niemal tak szybko, jak ich odpowiedniki napisane w asemblerze. Szczególnie wtedy, gdy tym programem był np. system operacyjny. Po dziś dzień C i jego nowszą, jeszcze bardziej elastyczną i uniwersalną wersję C++ wykorzystuje się tam, gdzie podstawowymi kryteriami są szybkość działania i objętość kodu wynikowego. Właśnie w C/C++ zostały napisane systemy operacyjne takie jak Unix i Linux, a nawet Windows. C i C++ ujawniają swoje możliwości w miarę zwiększania poziomu skomplikowania programu. Ich składnia została tak przemyślana, by umożliwić kompilatorom maksymalny poziom optymalizacji kodu. Często jednak, by osiągnąć najlepsze rezultaty, niezbędna jest wiedza na temat samego mikrokontrolera – takie połączenie umożliwia pisanie programów konkurujących z ich czysto asemblerowymi odpowiednikami. C++ wprowadza jeszcze większą swobodę działania, nawet w takich czynnościach jak deklaracje zmiennych. Udoskonalono i poszerzono o dodatkowe możliwości programowanie obiektowe, wprowadzając klasy i dziedziczenie. C i C++ wykorzystywany jest przez tych, którym zależy na wygodzie pracy i elastyczności języka przy jednoczesnych wysokich wymaganiach dotyczących kodu wynikowego. Dlatego też znakomita większość profesjonalnych programów została stworzona w C/C++ – od gier takich jak „Quake”, po potężne programy graficzne jak „3D Studio”. C++ jest standardem w programach *Open Source*, które „zapanowały” nad światem Linuksa. Wiele innych języków, takich jak Java, JavaScript, Perl czy PHP, jest opartych właśnie na C/C++. Nie oznacza to wcale, że język C (C++) nie nadaje się dla mikrokontrolerów. Wręcz przeciwnie, dzięki niemu można bez specjalnych zabiegów tworzyć szybki kod o niewielkiej objętości, zazwyczaj właśnie te dwa kryteria są najważniejsze. Podsumowując – Basic, wraz z całym IDE Bascom, wyposażonym w świetne narzędzia, takie jak symulator, programator i wiele innych, a także w biblioteki obsługujące bardzo szeroką gamę urządzeń, protokołów itd. jest naprawdę świetnym wyborem w sytuacji, gdy liczy się czas i prostota obsługi. Problem pojawia się wtedy, gdy ta prostota zaczyna być ograniczeniem, a także w przypadku, gdy zmieniamy platformę na cokolwiek innego niż AVR lub ‘51. C i C++ są dużo lepszym wyborem dla tych, którzy nie chcą się ograniczać tylko do tych dwóch rodzin mikrokontrolerów. A jeśli ktoś planuje kiedykolwiek zająć się programowaniem na PC lub nawet robieniem bardziej profesjonalnych stron www – znajomość C/C++ bardzo mu pomoże, a w wielu przypadkach umożliwi pokonanie przeszkód nie do pokonania dla innych języków.

Kuba Klimkiewicz