

## Przetwarzanie liczb zmiennoprzecinkowych część II.

# Magiczny świat liczb

W tym artykule Steven Goodwin zajmuje się problematyką przetwarzania liczb stałoprzecinkowych oraz sposobami wykorzystania ich w celu poprawienia wydajności aplikacji przeznaczonych dla urządzeń PDA. W artykule omówione zostały różnice między liczbami stało- i zmiennoprzecinkowymi, problemy związane z przenoszeniem oprogramowania między platformami oraz sposoby ich rozwiązywania. Do zrozumienia przykładów wymagana jest podstawowa teoretyczna wiedza matematyczna i znajomość języka C.

STEVEN GOODWIN

**W**iele algorytmów, zwłaszcza tych używanych w nauce i projektach multimedialnych, wymaga zastosowania liczb zmiennoprzecinkowych. To znaczy, liczb posiadających miejsca dziesiętne (po przecinku), takich jak 9,2 lub 0,65. My ludzie, używamy ich tak jak innych liczb. Wykonujemy na nich działania matematyczne w taki sam sposób, jak na liczbach 92 lub 65 – po zakończeniu obliczeń wstawiamy przecinek na odpowiedniej pozycji.

W przypadku komputerów nie jest to takie łatwe. Począwszy od procesora i486DX, wszystkie procesory produkowane przez firmę Intel mają koprocesor arytmetyczny FPU (floating-point unit) wykonujący bardzo szybko obliczenia zmiennoprzecinkowe. Koprocesory te zużywają jednak wiele energii i w rezultacie nie są stosowane w urządzeniach przenośnych, takich jak np. PDA.

Koprocesor arytmetyczny nie jest niezbędny dla edytorów tekstowych i klientów HTML,

ale odtwarzacze MP3/Ogg oraz aplikacje graficzne (zwłaszcza pracujące w trybie 3D) wymagają wykonywania wielu obliczeń zmiennoprzecinkowych. Powoduje to znaczne spowolnienie pracy urządzeń przenośnych z zainstalowanym Linuxem, gdyż procesor musi wykonywać te obliczenia, emulując FPU.

Istnieje jednak inne rozwiązanie – części każdego algorytmu można napisać w taki sposób, aby obliczenia były wykonywane na liczbach stałoprzecinkowych. Liczby te są traktowane przez procesor jak liczby całkowite (tzn. bez miejsc dziesiętnych) i mogą być przetwarzane bezpośrednio, bez emulacji. Dzięki temu można znacznie przyspieszyć działanie aplikacji.

## Kłopotliwy przecinek

Podczas wykonywania obliczeń arytmetycznych na papierze możemy swobodnie przenosić przecinek, zależnie od naszych potrzeb. Czasami więcej cyfr znajduje się przed, czasami po przecinku. Robimy to automatycznie, kiedy zbliżamy się do brzegu kartki przenosimy cyfry na początek nowego wiersza i kontynuujemy obliczenia. Przecinek zmienia wtedy swoje miejsce, ułatwiając nam tym samym obliczenia. Powyższy model obliczeń nazywany jest zmiennoprzecinkowym.



```
float fValue = 14.5f;
/* Deklaracja zmiennej
zmiennoprzecinkowej w C. */
```

W pamięci komputera każda liczba zmiennoprzecinkowa jest przechowywana w formacie zgodnym ze standardem IEEE 754. Dla liczb pojedynczej precyzji (zapisanych w formacie 32-bitowym) format ten opisano szczegółowo w Tabeli 1.

Obie połowy liczby (E i M) nie odnoszą się do wartości znajdujących się przed i po przecinku. Mantysa (mantissa, M) odnosi się do całej liczby (z usuniętym przecinkiem). Liczba miejsc dziesiętnych, o którą należy przesunąć przecinek oraz kierunek, w którym należy to zrobić w celu przywrócenia pierwotnej liczby, jest określana jako wykładnik (exponent, E). Liczbę tę można następnie wyprowadzić, tak jakby była zapisana w notacji naukowej:

$$N = -1^S * 1. M * B^E$$

S oznacza znak, M to znormalizowana mantysa, B to podstawa, E wykładnik. Na przykład:

```
Znak: 0 (liczba dodatnia)
Wykładnik: -1
        (podziel przez 10)
```

Mantysa: 145

$$N = 145 * 10^{-1} = 14,5$$

Jest to oczywiście dość pobieżne omówienie tego standardu. W standardzie IEEE uwzględniono wiele dodatkowych pomysłów (np. normalizacja czy też przesunięcie) i wprowadzono specjalne wzorce bitowe umożliwiające reprezentowanie wartości, takich jak nieskończoność oraz NaN (Not a number), czyli nie dających się wyrazić liczbami. Jeżeli chcesz dokładniej zapoznać się z zagadnieniem liczb zmiennoprzecinkowych, przeczytaj inne artykuły ([1] oraz [2]) na ten temat dostępne w sieci.

Zaletą notacji zmiennoprzecinkowej jest możliwość pracy z bardzo dużymi liczbami (do  $3,402823 * 10^{38}$ ), ale ma ona też wady – nie można w niej dokładnie przedstawić wartości niektórych liczb. Powyższy fakt może się okazać równie dobrze drobnym szczegółem, jak i poważną wadą, zależnie od problemu, który należy rozwiązać. Jeżeli mierzysz średnicę systemu słonecznego, dodanie do wyniku wartości rzędu średnicy atomu nie będzie miało wpływu na wynik Twoich obliczeń. Jeżeli do tych obliczeń wykorzystamy zapis zmiennoprzecinkowy, dokładność obliczeń nie uwzględni w wyniku różnicy powodowanej przez dodanie tak małej wartości. Jeżeli jednak pracujesz na wartościach rzędu atomów, dodanie wartości jednego atomu będzie miało znaczący wpływ na wynik obliczeń, ponieważ ich wykładniki mają zbliżoną wartość.

Innym mankamentem notacji zmiennoprzecinkowej jest nieprecyzyjna reprezentacja małych liczb, przykładowo 2,8 jest w pamięci widziana jako 2,799999999999. Dzieje się tak, ponieważ liczba 1,4 może mieć podstawę równą 10 (na której opieramy nasze obliczenia), ale nierówną 2 (na której opiera swoje obliczenia komputer). Na załączonym przykładzie można zobaczyć, jak zgodnie ze standardem IEEE, liczba 1,4 będzie reprezentowana w pamięci komputera. Jeżeli zakres liczb zmiennoprzecinkowych okaże się niewystarczający, można użyć zmiennych typu double lub double-extended.

Jak można łatwo się domyśleć, obliczenia arytmetyczne na dwóch liczbach zapisanych w tym standardzie wymagają wiele zachodu. FPU wykonuje standardowe obliczenia ary-

### Ramka 1. Reprezentacja cyfry 14.5 w postaci binarnej

```

1      111111
... 2631  ----- ...
   84268421.248163264
-----
... 00001110.100000 ...

```

metyczne, ale oprócz tego musi również uporać się z różnymi wykładnikami, przepelnionymi mantysami, bitami znaku oraz oceną nowego wykładnika.

### Uniwersalny zapis binarny

Notacja stałoprzecinkowa opiera się na określaniu stałej liczby bitów w całym zakresie liczb, na których są zapisywane liczby przed i po przecinku. Przykładowo część całkowita może być zawsze zapisywana na 12 bitach, a ułamkowa na 20. (Patrz: Ramka 2). Ogranicza to znacznie dostępny zakres liczb całkowitych, ale oznacza jednocześnie większą precyzję, ponieważ każda liczba z tego zakresu może być reprezentowana w powyższej notacji.

Liczbę 14,5 można przedstawić w następujący sposób: liczbę całkowitą 14 zapisujemy w systemie dwójkowym na 12 bitach, gdzie mantysa wynosi 000000001110. Część ułamkową, czyli 0,5, zapisujemy w systemie dwójkowym na 20 bitach: 10000000000000000000. W ten sposób można zapisać całą liczbę w systemie dwójkowym: 00000000111010000000000000000000. Jeżeli wynik ten zostanie wyświetlony jako liczba całkowita, na ekranie zobaczymy liczbę 15204352!

Odczytywanie części ułamkowej liczb stałoprzecinkowych zapisanych w systemie dwójkowym nie jest trudniejsze, niż odczytywanie liczb całkowitych zapisanych w systemie dwójkowym. Wystarczy zastosować metodę potęgowej liczby dwa dla miejsc po przecinku, tak jak to przedstawiono w Ramce 1.

Obliczenia można wykonywać w takim przypadku na liczbach całkowitych, zatem FPU staje się zbędny. Składnik 0,5 jest zapisywany w konwencji 2.19, tak że wartości 0,5+0,5 automatycznie utworzą wartość 1,0 i zostaną zapisane na bitach przeznaczonych dla części całkowitej.

W tym przypadku nie zadeklarowaliśmy jawnie bitu znaku. Nie trzeba tego robić, ponieważ część całkowita zostanie binarnie zapisana jako uzupełnienie dwójkowe (podobnie jak zwykle liczby całkowite). Jest to zgodne ze sposobem przetwarzania liczb przez CPU, co w rezultacie umożliwia zaoszczędzenie czasu procesora. Zakres wartości, który można zapisać przy użyciu naszej zdefiniowanej konwencji 12.20, mieści się w przedziale od -2048 do +2047 (-211 do 211-1).

Tryb stałoprzecinkowy może oznaczać w przypadku większości aplikacji ograniczenia. My jednak szukamy rozwiązania dla określonego problemu i przyjmujemy ogra-

### Ramka 2. Jak to podzielić?

Podział 12.20 jest arbitralny, ponieważ działania na liczbach stałoprzecinkowych są analogiczne do działań na liczbach całkowitych. Na podstawie stopnia złożoności problemu oraz możliwości obliczeniowych sprzętu możesz samodzielnie określić, ile bitów będzie przeznaczonych na zapis części ułamkowej, a ile części całkowitej.

Rozważając tę kwestię, zastanów się, w jakim zakresie wartości liczby będą się mieścić. Przykładowo zajmujesz się przetwarzaniem sygnałów o przebiegach sinusoidalnych. Zatem wszystkie wartości pojedynczej przetwarzanej funkcji będą mieścić się w zakresie od -1 do +1. Przy założeniu przetwarzania (sumowania) nie więcej niż 16 przebiegów sinusoidalnych, zakres +/- 16 wydaje się być w zupełności wystarczający. Możemy więc założyć konwencję zapisu 6.26.

Dzięki temu możesz uzyskać większą precyzję w porównaniu do standardowych obliczeń na liczbach zmiennoprzecinkowych.

Kwestia sprzętu: korzystam z 32-bitowego komputera (opartego na procesorze typszeregu i86), jako popularnej dobrze udokumentowanej platformy sprzętowej, więc w moim przypadku dobrym pomysłem jest użycie wszystkich 32 bitów do zapisu liczb stałoprzecinkowych. Jednakże dużo mniejsze systemy mogą dysponować jedynie procesorami 16-bitowymi, więc musielibyśmy dopasować kod do ich możliwości. Liczby stałoprzecinkowe są zwykle używane w celu zwiększenia prędkości oprogramowania oraz poprawy gospodarki pamięcią. Rozwiązania stałoprzecinkowe są dostosowane znacznie bardziej do określonej architektury niż inne oprogramowanie i z tego powodu najlepszym rozwiązaniem jest tworzenie algorytmów stałoprzecinkowych zorientowanych na konkretną platformę sprzętową. Rozwiązania te powinny być jasno udokumentowane i oznaczone stosownymi etykietami.

### Tabela 1. Format IEEE 754

Bit:	31	30-23	22-0
Description:	Sign bit	Exponent	Mantissa
	s	eeeeeeee	mmmmmmmmmmmmmmmmmmmmmmmmmm

niczenia tego trybu, wykorzystując jednocześnie jego zalety – szybkość i precyzję.

## Nie komplikujmy sprawy

Liczby stałoprzecinkowe są przechowywane w pamięci jako liczby całkowite, zwykle typu `int` lub `long`. Aby móc korzystać z większego zakresu liczb, można użyć typu danych `long long` lub dwóch zmiennych – oznacza to oczywiście konieczność dodatkowego przetwarzania przy przechodzeniu między tymi dwoma zmiennymi. Ponieważ naszym celem jest zmniejszenie ilości danych, które trzeba przetworzyć, ta metoda nie przyda nam się na wiele.

## I zabieramy się do pracy

Niezależnie od tego, czy piszesz nową procedurę w notacji stałoprzecinkowej, czy też dopasowujesz kod do istniejącej już aplikacji, musisz mieć solidne podstawy do pracy. Tą podstawą będzie dla nas plik nagłówkowy `fixed.h` zawierający wszystkie niezbędne makra oraz prototypy funkcji, które będą następnie wykorzystywane w kodzie. Należy również utworzyć pomocniczy plik `fixed.c`, tak aby korzystanie z funkcji było możliwe. Mimo iż ostatecznie kod będzie oparty wyłącznie na makrach (ze względu na prędkość), plik źródłowy w C ułatwi znacznie proces tworzenia aplikacji (łącznie z debugowaniem). Może on również zawierać funkcje umożliwiające tworzenie i usuwanie tymczasowych obszarów roboczych, które są czasami tworzone przez bibliotekę stałoprzecinkową. Przykłady pojawiają się w dalszej części artykułu.

Istnieje wiele różnych składników biblioteki tego typu. Na przykład:

- Procedury konwersji z notacji stało- na zmiennoprzecinkową i odwrotnie.
- Podstawowe działania matematyczne: dodawanie, odejmowanie, mnożenie i dzielenie.
- Stałe matematyczne, takie jak  $\pi$  lub  $e$ .
- Funkcje matematyczne, takie jak sinus lub cosinus.

## Java

W artykule przykłady zostały napisane w C, tak by były zrozumiałe. Programiści pracujący w Javie mogą zapoznać się z podobnym kodem pod adresem [3]. Dekoder Ogg Vorbis, który zainspirował mnie do napisania tego artykułu, wystarcza w zupełności do rozwiązywania konwencjonalnych problemów związanych z liczbami stałoprzecinkowymi. Dekoder ten można znaleźć pod adresem [4]. Wielkie dzięki dla Richarda Cohena za udostępnienie źródła!

## Ramka 3. C++

Programiści pracujący w C++ mają do dyspozycji funkcję znaną jako przeciążanie operatora. Umożliwia ona zmianę sposobu pracy podstawowych operatorów (takich jak dodawanie, odejmowanie itd.) w odniesieniu do klas. Umożliwia to stworzenie klasy o nazwie `CFixedPoint` i zrezygnowanie z makr. Dzięki temu elementy stałoprzecinkowe kodu są czytelniejsze, a algorytm bardziej przejrzysty.

Jeżeli język programowania, którego używasz, oferuje taką funkcję, rozważ jej zastosowanie.

- Różne funkcje pomocnicze, takie jak zaokrąglanie.

Na pierwszy rzut oka może się wydawać, że czeka nas wiele pracy. Nic bardziej mylnego. Konwertujemy jedynie niewielkie części aplikacji, musimy zatem napisać ponownie (lub skopiować z innego źródła) jedynie część z powyższych składników. Nawet dzielenie, tak często przez nas używane, może okazać się całkowicie zbędne w tym przypadku!

Podstawowy plik nagłówkowy powinien zawierać wszystkie pozostałe informacje o zastosowaniu kodu. Rozsądnym rozwiązaniem jest unikanie magicznych liczb 12 i 20 przy opisywaniu liczby bitów używanych w naszej notacji, ponieważ mogą się one później zmienić. Należy również zdefiniować typ danych dla liczb, aby określić odpowiednią liczbę bi-

tów dla części całkowitych i ułamkowych.

```
typedef int FIXP;
/* To musi być liczba zapisana
na 32 bitach */

#define FIX_FRACBITS 20
#define FIX_INTBITS 12
```

Jak łatwo zauważyć, nazwy wszystkich makr rozpoczynają się prefiksem `FIX_`. Także każdy składnik będzie definiowany określoną liczbą (12 & 20), mimo iż w praktyce rozmiar składnika można ustalić w następujący sposób:

```
#define FIX_INTBITS ( sizeof ( U
FIXP ) *8 ) -FIX_FRACBITS)
```

Każda z powyższych metod jest dobra. Kompilator optymalizuje makro podczas generowania kodu binarnego, zatem nie spowolni ono działania aplikacji w tych nielicznych przypadkach, kiedy będzie potrzebne. Należy również rozważyć możliwość włączenia bitu znaku do makra `FIX_INTBITS`.

W praktyce jednym z największych problemów w przypadku oprogramowania, niezależnie od jego rodzaju, jest testowanie. W jaki sposób przeprowadzić testy? Czy możemy łatwo wykryć błędy? Czasami bardzo trudno jest odpowiedzieć na takie pytania.

W przypadku liczb stałoprzecinkowych określenie liczby bitów na potrzeby zapisu części całkowitej i ułamkowej jest trudniej-

## Listing 1. Sprawdzanie drzewa

```
01 FIXP fixFloatToFixed(float fValue)
02 {
03     FIXP Unity, Result;
04
05     if ((int)fValue >= (1<<(FIX_INTBITS-1))) /* -1 because of sign
bit */
06         fprintf(stderr, "FIXP: Integral overflow of %f\n", fValue);
07
08     if ((int)fValue < -(1<<(FIX_INTBITS-1))) /* -1 because of sign
bit */
09         fprintf(stderr, "FIXP: Negative integral overflow of %f\n", fValue);
10
11     if (fValue < 1.0f/(1<<FIX_FRACBITS) && fValue > -1.0f/(1<<FIX_FRACBITS)
&& fValue != 0)
12         fprintf(stderr, "FIXP: Fractional underflow of %f\n", fValue);
13
14     Unity = 1<<FIX_FRACBITS;
15     Result = (FIXP) (Unity * fValue);
16
17     return Result;
18 }
```

sze, niż mogłoby się wydawać. Liczby często przekraczają ustalony przez nas zakres, generując w ten sposób bez ostrzeżenia wiele błędów. Sprawdzanie wartości każdej liczby na każdym etapie procesu w celu uniknięcia błędów mija się z celem, gdyż aplikacja działałaby wtedy bardzo wolno. Dobrym pomysłem jest w takiej sytuacji użycie w odpowiednim miejscu makra reprezentującego algorytm lub wywołanie osobnej funkcji.

Można to zrobić przy pomocy GCC, kompilując kod z flagą DDEBUG i przełączając się między dwoma wersjami kodu:

```
#ifndef DEBUG
#define FLOAT_TO_FIXED (__f) >
fixFloatToFixed (__f)
#else
#define FLOAT_TO_FIXED (__f) >
(FIXP) ( (__f) * >
(1<FIX_FRACBITS))
#endif
```

Funkcja fixFloatToFixed może również sprawdzać zakres obu składników, wyszukiwać błędy lub nawet tworzyć statystykę wystąpień najczęściej używanych bitów dla części całkowitych i ułamkowych, co byłoby trudne do wykonania w przypadku makra. Po zapoznaniu się z kilkoma podstawowymi pojęciami możemy przystąpić do dalszej pracy.

## Konwertowanie

W tej kategorii niezbędne będą co najmniej dwie funkcje – jedna konwertująca liczby zmiennoprzecinkowe na stałoprzecinkowe, a druga odpowiednio liczby stałoprzecinkowe na zmiennoprzecinkowe. Możemy również stworzyć inne funkcje, konwertujące przykładowo zapis oparty na dwóch liczbach całkowitych na zapis stałoprzecinkowy. W tym przypadku jednak ograniczymy się do zaimplementowania tylko funkcji potrzebnych nam do wykonania zadania. Przed przekonwertowaniem liczb do notacji stałoprzecinkowej, należy sprawdzić trzy rzeczy:

## Ramka 4. Przesunięcia bitowe

Jedną z najczęściej spotykanych operacji przesunięcia są rozmaite postacie  $1 < n$  (gdzie w naszym przypadku  $n=20$ ). Operacje te są używane w dwóch przypadkach: do zapisania wartości 1.0 w notacji stałoprzecinkowej, którą można następnie wykorzystać do pojedynczego mnożenia przy konwersjach liczb zmiennoprzecinkowych, oraz w przypadku wyrażeń  $(1 < n) - 1$ , gdy każdy bit wyniku w części ułamkowej jest ustawiony na 1. W ten sposób tworzona jest idealna maska bitowa umożliwiająca oddzielenie części ułamkowej, tak jak ma to miejsce w przypadku funkcji fixRound.

1. Czy liczby nie są zbyt duże (np. większe niż 2047)?
2. Czy liczby nie są zbyt małe (np. mniejsze niż -2048)?
3. Czy część ułamkowa może być reprezentowana przy użyciu dostępnej liczby bitów?

Listing 1 pokazuje kod wykonujący trzy powyższe operacje. Podczas konwertowania z powrotem na liczby zmiennoprzecinkowe (patrz Listing 2) nie będziemy sprawdzać zakresu ani dokładności. Jest to możliwe, chociaż dość trudne, ale możemy z tego zrezygnować, ponieważ nie wnosi dla nas nic istotnego.

Jeżeli ostatnią funkcję zapiszemy przy użyciu makra, możemy wtedy zrezygnować z jawnej konwersji typu danych, takiej jak poniższa:

```
#define FIXED_TO_FLOAT (__i) >
( ( (__i) >FIXP_FRACBITS) >
+ (float) ( ( (__i) & >
( (1<FIXP_FRACBITS) -1)) >
/ (1<FIXP_FRACBITS))
```

Zapis części całkowitej i ułamkowej liczby w jednej zmiennej może powodować problemy z logiką zapisu. Logika sama w sobie nie stanowi jednak problemu, ale sposób jej zaimplementowania w makrach jest nieco skomplikowany.

Wszystko to sprowadza się do kilku różnych kombinacji przesunięć i masek bitowych (patrz Ramka 4: Przesunięcia bitowe).

Następnie można przetestować funkcje konwertujące przy użyciu prostego programu w C:

```
FIXP fixed;
float not_fixed;
fixed = fixFloatToFixed (14.5f);
printf ('14.5 in decimal >
= %d? \n', fixed);
not_fixed >
= fixFixedToFloat (fixed);
printf ('Does 14.5 = %f? \n', >
not_fixed);
```

## I na koniec

We wszystkich podanych przykładach założono, że każda liczba stałoprzecinkowa jest zapisana w formacie 12.20. Może oczywiście tak być. Jednak część oprogramowania wymaga zastosowania dwóch (lub wielu) różnych formatów zapisu w danym programie. Kiepskim rozwiązaniem jest tworzenie dwóch nagłówków z różnymi nazwami tylko w celu zapewnienia większej precyzji w określonych krytycznych obszarach programu. Lepiej jest pracować w jednym domyślnym formacie i utworzyć ogólne makra, z których mogą korzystać biblioteka oraz główny program.

```
#define FLOAT_TO_FIXED_GEN >
(__f, __fracbits) (FIXP) >
( (__f) * (1< (__fracbits)))
#define FLOAT_TO_FIXED (__f) >
FLOAT_TO_FIXED_GEN >
(__f, FIX_FRACBITS)
```

W tym artykule zajmowaliśmy się tworzeniem, przetwarzaniem oraz implementowaniem algorytmów opartych na liczbach stałoprzecinkowych. W poprzednim numerze (Linux Magazine 1/2004) przedstawiliśmy niejako pierwszą część artykułu poświęconego stosowaniu liczb zmiennoprzecinkowych. ■

## Listing 2. Konwersja do typu zmiennoprzecinkowego

```
01 float fixFixedToFloat(FIXP          & ((1<<FIX_FRACBITS)-1)); /*
   iValue)                          isolate fraction */
02 {                                  09   fFraction /= (float)
03 float Result;                      (1<<FIX_FRACBITS);
04 float fIntegral, fFraction;         10
05                                     11   Result = fIntegral +
06   fIntegral = (float)               fFraction;
   (iValue>>FIX_FRACBITS);            12
07                                     13   return Result;
08   fFraction = (float) (iValue       14 }
```

## INFO

- [1] Opis standardu dla binarnej arytmetyki zmiennoprzecinkowej: <http://grouper.ieee.org/groups/754>
- [2] IEEE Floating Point Standard: [http://www.tutorgig.com/encyclopedia/getdefn.jsp?query=IEEE\\_754](http://www.tutorgig.com/encyclopedia/getdefn.jsp?query=IEEE_754)
- [3] Java versions: <http://www.ai.mit.edu/people/hqml/imode/fplib/FP.java>
- [4] Ogg Vorbis decoder: <http://lorien.handhelds.org/ftp.arm.linux.org.uk/people/nico/vorbis>