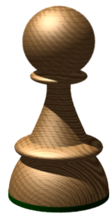
Pawn



embedded scripting language

Fixed Point Support

Contents

January 2015 CompuPhase

Eerste Industriestraat 19–21, 1401VL Bussum The Netherlands
telephone: (+31)-(0)35 6939 261
e-mail: info@compuphase.com
WWW: http://www.compuphase.com

Typeset with TEX in the “DejaVu” typeface family.

Introduction

The "PAWN" programming language is a simple C-like extension/scripting language. The only data type that it supports is a 32-bit integer, called a *cell*. The PAWN programming language is described in its manual and it is freely available; see the section "Resources" for more information. This Fixed Point support library adds "fixed point decimal arithmetic" and fixed point rational values to the "PAWN" programming language.

In computer applications, rational values have limited precision, regardless of how they are implemented. The advantage of fixed point arithmetic over the more common floating point arithmetic is that fixed point defines the precision in an exact and intuitive manner.

It is well known, for example, that the value 0.1 cannot be represented exactly in the floating point format standardized in IEEE 754* —which is the ubiquitous floating point format. This imperfection leads to the following PAWN program to give a surprising result:

LISTING: floating point arithmetic example, giving a surprising result

```
#include <float>

main()
    {
    new Float: a = 0.0
    new Float: b = 1.0

    for (new i = 0; i < 10; i++)
        a += 0.1

    if (a == b)
        printf("%f and %f are equal\n", a, b)
    else
        printf("%f is not the same as %f\n", a, b)
    }
```

The above program is in PAWN, but the phenomenon is shared by all languages that use the IEEE 754 floating point format.†

* IEEE 754 implements *binary* floating point and the value 0.1 is an infinitely recurring fraction in binary base.

† I have verified this with a similar program in C, using both the single-precision and double-precision floating point formats (see "Resources" for the source code). A tutorial for the Python language also mentions this pitfall and amplifies that it is "*not a bug*" in Python.

However, if the program were translated to use the fixed point library instead of the floating point library, it would give the intuitive (and correct!) answer that ten times 0.1 is indeed 1.0.

The above program shows that tiny errors in floating point values can accumulate to larger, noticeable errors after repeated operations. Sometimes, the intermediate incorrectness is hidden by extra "guard digits" in internal CPU registers and rounding in library functions. However, even single operations can cause errors. From the "Decimal Arithmetic FAQ, Part 1" (see section "Resources") comes the following example:

> Consider the calculation of a 5% sales tax on an item (such as a $0.70 telephone call), which is then rounded to the nearest cent. Using double precision floating-point, the result of $0.70 × 1.05 is 0.73499999999999998667732370449812151491641998291015625; the result should have been 0.735 (which would be rounded up to $0.74) but instead the rounded result would be $0.73

Fixed point arithmetic, and especially *decimal* arithmetic is appropriate for applications where the precision is important and all numbers lie in a limited range; commercial and financial applications are a prime example, but graphic/programs also benefit from fixed point decimal arithmetic.

The fixed point format used in this library uses three decimal digits and stores the values in two's complement. This gives a range of -2147483 to +2147482 with 3 digits behind the decimal point. Fixed point arithmetic also goes by the name "scaled integer" arithmetic. Basically, a fixed point number is the numerator of a fraction where the denominator is implied. For this library, the denominator is 1000 —therefore, the integer value 12345 stands for $\frac{12345}{1000}$ or 12.345.

In rounding behaviour, however, there is a subtle difference between fixed point arithmetic and straight-forward scaled integer arithmetic: in fixed point arithmetic, it is usually intended that the least significant digit should be rounded before any subsequent digits are discarded; but many scaled integer arithmetic implementations just "drop" any excess digits. In other words, $^2/_3$ in fixed point arithmetic results in 0.667, which is more accurate than the scaled integer result of 0.666.

Next to the decimal arithmetic model (used by this library), there are also fixed point packages that use a *binary model*, where the fractional part is specified in the number of "bits" rather than "digits". The binary model has significantly different properties than the decimal model, and it is used for entirely different purposes. In discussions on fixed point arithmetic (e.g. versus floating-point arithmetic), one should be careful about the radix of the fixed point format that is discussed. As a side note, fixed point decimal arithmetic is sometimes (incorrectly) referred to as “BCD” arithmetic —the *Binary Coded Decimals* format stores one digit in each nibble (half byte, or four bits). Fixed-point arithmetic and BCD format may be combined, but the two are not the same. As already noted, this library uses two’s complement rather than BCD.

Implementing the library

The fixed point support library consists of the files FIXED.C and FIXED.INC. The C file may be "linked in" to a project that also includes the PAWN Abstract Machine (AMX.C), or it may be compiled into a DLL (Microsoft Windows) or a shared library (Linux). The .INC file contains the definitions for the PAWN compiler of the native functions in FIXED.C, as well as several user-defined operators. In your PAWN programs, you may either include this file explicitly, using the `#include` preprocessor directive, or add it to the "prefix file" for automatic inclusion into any PAWN program that is compiled.

The FIXED.INC also sets the rational number format for the PAWN compiler to a fixed point number with three decimal digits (using `#pragma rational`). This may lead to a conflict if a different rational number format was already set. Specifically, you may not be able to use this fixed point extension module together with a floating point module. Such conflicts can be resolved by removing the `#pragma rational` directive from either module.

The "Implementer's Guide" for the PAWN toolkit gives details for implementing the extension module described in this application note into a host application. The initialization function, for registering the native functions to an abstract machine, is `amx_FixedInit` and the "clean-up" function is `amx_FixedCleanup`. In the current implementation, calling the clean-up function is not required.

If the host application supports dynamically loadable extension modules, you may alternatively compile the C source file as a DLL or shared library. No explicit initialization or clean-up is then required. Again, see the Implementer's Guide for details.

The extension module AMXCONS.C (console input/output) has support for fixed point values. You have to enable this support by compiling the file with the `FIXEDPOINT` macro defined.

Usage

Depending on the configuration of the PAWN compiler, you may need to explicitly include the FIXED.INC definition file. To do so, insert the following line at the top of each script:

```
#include <fixed>
```

The `#pragma rational` setting in FIXED.INC allows you to specify rational literal numbers directly. For example:

```
new Fixed: amount = 123.45
amount += 78.90
```

To convert from integers to fixed point values, use one of the functions `fixed` or `strfixed`. The function `fixed` creates a fixed point number with the same integral value as the input value and a fractional part of zero. Function `strfixed` makes a fixed point number from a string, which can include a fractional part.

A user-defined assignment operator is implemented to automatically coerce integer values on the right hand to a fixed point format on the left hand. That is, the lines:

```
new a = 10
new Fixed: b = a
```

are equivalent to:

```
new a = 10
new Fixed: b = fixed(a)
```

To convert back from fixed point numbers to integers, use the functions `fround` and `ffract`. Function `fround` is able to round upwards, to round downwards, to "truncate" and to round to the nearest integer. Function `ffract` gives the fractional part of a fixed point number, but still stores this as a fixed point number.

The common arithmetic operators: `+`, `-`, `*` and `/` are all valid on fixed point numbers, as are the comparison operators and the `++` and `--` operators. The modulus operator `%` is forbidden on fixed point values.

The arithmetic operators also allow integer operands on either left/right hand. Therefore, you can add an integer to a fixed point number (the result will be a fixed point number). This also holds for the comparison operators: you can compare a fixed point number directly to an integer number (the return value will be `true` or `false`).

Native functions

fabs

Return the absolute value of a fixed point number

Syntax: `Fixed: fabs(Fixed: value)`

`value` The value to return the absolute value of.

Returns: The absolute value of the parameter.

fdiv

Divide a fixed point number

Syntax: `Fixed: fdiv(Fixed: oper1, Fixed: oper2)`

`oper1` The numerator of the quotient.

`oper2` The denominator of the quotient.

Returns: The result: `oper1`/`oper2`.

Notes: The user-defined `/` operator forwards to this function.

See also: `fmul`

ffract

Return the fractional part of a number

Syntax: `Fixed: ffract(Fixed: value)`

`value` The number to extract the fractional part of.

Returns: The fractional part of the parameter, in fixed point format. For example, if the input value is “3.14”, `ffract` returns “0.14”.

See also: `fround`

fixed

Convert integer to fixed point

Syntax: `Fixed: fixed(value)`

`value` the input value.

Returns: A fixed point number with the same (integral) value as the parameter (provided that the integral value is in range).

See also: `fround`, `strfixed`

fmul

Multiply two fixed point numbers

Syntax: `Fixed: fmul(Fixed: oper1, Fixed: oper2)`

`oper1`
`oper2` The two operands to multiply.

Returns: The result: `oper1` $\times$ `oper2`.

Notes: The user-defined `*` operator forwards to this function.

See also: `fdiv`

fmuldiv

Fixed point multiply followed by a divide

Syntax:

```
Fixed: fmuldiv(Fixed: oper1, Fixed: oper2,
               Fixed: divisor)
```

`oper1`
`oper2` The two operands to multiply (before the divide).

`divisor` The value to divide `oper1` $\times$ `oper2` by.

Returns: The result: $\frac{oper1 \times oper2}{divisor}$.

Notes: This function multiplies two fixed point numbers and then divides it by a third number ("`divisor`"). Since it avoids rounding the intermediate result (the multiplication), the result of `fmuldiv(a, b, c)` may have higher precision than "`(a * b) / c`".

See also: `fdiv`, `fmul`

fpower

Raise a fixed point number to a power

Syntax: `Fixed: fpower(Fixed: value, exponent)`

- `value` — The value to raise to a power; this is a fixed point number.
- `exponent` — The exponent is a whole number (integer). The exponent may be zero or negative.

Returns: The result: $value^{exponent}$; this is a fixed point value.

Notes: For exponents higher than 2 and fractional values, the `fpower` function may have higher precision than repeated multiplication, because the function tries to calculate with an extra digit. That is, the result of `fpower(3.142, 4)` is probably more accurate than `3.142 * 3.142 * 3.142 * 3.142`.

See also: `fsqroot`

fround

Round a fixed point number to an integer value

Syntax:

```
fround(Fixed: value,
       fround_method: method=fround_round)
```

- `value` — The value to round.
- `method` — The rounding method may be one of:
  - `fround_round` — (default) round to the nearest integer, a fractional part of exactly 0.5 rounds upwards;
  - `fround_floor` — round downwards;
  - `fround_ceil` — round upwards;
  - `fround_tozero` — round downwards for positive values and upwards for negative values (“truncate”);

`fround_unbiased`
round to the nearest *even* integer number when the fractional part is exactly 0.5 (the values "1.5" and "2.5" both round to "2"). It is also known as "Banker's rounding".

Returns: The rounded value, as an integer (an untagged cell).

Notes: When rounding negative values upwards or downwards, note that -2 is considered smaller than -1.

See also: `ffract`

---

fsqroot

Return the square root of a value

Syntax: `Fixed: fsqroot(Fixed: value)`

`value` The value to calculate the square root of.

Returns: The result: the square root of the input number.

Notes: This function raises a "domain" error is the input value is negative.

See also: `fpower`

---

strfixed

Convert from text (string) to fixed point

Syntax: `Fixed: strfixed(const string[])`

`string` The string containing a number in characters. This may be either a packed or unpacked string. The string may have a fractional part, e.g. "`123.45`".

Returns: The value in the string, or zero if the string did not start with a valid number.

Custom operators

All custom operators are declared “native” or “stock”. Operators that you do not use in your script take no space in the P-code file.

`Fixed:operator*(Fixed:oper1, Fixed:oper2)`

`Fixed:operator/(Fixed:oper1, Fixed:oper2)`

`Fixed:operator=(oper)`

`Fixed:operator++(Fixed:oper)`

`Fixed:operator--(Fixed:oper)`

`Fixed:operator*(Fixed:oper1, oper2)` (*“*” is commutative*)

`Fixed:operator/(Fixed:oper1, oper2)`

`Fixed:operator/(oper1, Fixed:oper2)`

`Fixed:operator+(Fixed:oper1, oper2)` (*“+” is commutative*)

`Fixed:operator-(Fixed:oper1, oper2)`

`Fixed:operator-(oper1, Fixed:oper2)`

`bool:operator>(Fixed:oper1, oper2)`

`bool:operator>(oper1, Fixed:oper2)`

`bool:operator>=(Fixed:oper1, oper2)`

`bool:operator>=(oper1, Fixed:oper2)`

`bool:operator<(Fixed:oper1, oper2)`

`bool:operator<(oper1, Fixed:oper2)`

`bool:operator<=(Fixed:oper1, oper2)`

`bool:operator<=(oper1, Fixed:oper2)`

`bool:operator==(Fixed:oper1, oper2)` (*“==” is commutative*)

`bool:operator!=(Fixed:oper1, oper2)` (*“!=” is commutative*)

Resources

The PAWN toolkit can be obtained from www.compuphase.com in various formats (binaries and source code archives). The manuals for usage of the language and implementation guides are also available on the site in Adobe Acrobat format (PDF files).

The limitations of IEEE 754 floating point arithmetic are well documented, but not very widely known. An introductory article on the pitfalls of floating point arithmetic is "The Perils of Floating Point" by Bruce M. Bush, available on www.lahey.com/float.htm. A more in-depth article, which is the source of one of the examples in this application note, is IBM's "Decimal Arithmetic FAQ, Part 1" at www2.hursley.ibm.com/decimal/decifaq1.html.

The source code in C that demonstrates the limited precision of IEEE 754 floating point arithmetic (using double precision floating point) is below. See page 1 for a description of the problem. The "`volatile`" variable attribute is to avoid the compiler from "hiding" the error from view: an optimizing C/C++ compiler may keep the accumulated result in an internal processor register and avoid the incremental truncation error —a highly optimizing C/C++ compiler might even detect the pattern and set "a" to 1.0 without looping at all.

LISTING: floating point surprise, in C

```
#include <stdio.h>

{
    volatile double a = 0.0;
    volatile double b = 1.0;

    int i;
    for (i = 0; i < 10; i++)
        a += 0.1;

    if (a == b)
        printf("%f and %f are equal\n", a, b);
    else
        printf("%f is not the same as %f\n", a, b);

    return 0;
}
```

Index

- ◇ Names of persons (not products) are in *italics*.
- ◇ Function names, constants and compiler reserved words are in `typewriter font`.