Pawn

embedded scripting language

Floating Point Support Library

August 2005

Abstract

The "PAWN" programming language has only native support for 32-bit integers. This document describes an extension to the PAWN run-time environment that adds support for "floating point" rational values.

The software that is associated with this application note can be obtained from the company homepage, see section "Resources"

Typeset with TEX in the "Computer Modern" and "Palatino" typefaces at a base size of 11 points.

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 Floating Point support library adds "IEEE 754 floating point" arithmetic and values to the "PAWN" programming language.

Floating point values represent very small values and very large values with (approximately) the same number of significant digits. This property makes floating point numbers very suitable for engineering and general-purpose arithmetic with rational values. For the IEEE 754 32-bit format, the number of significant digits is about 7.

In computer applications, rational values have limited precision, regardless of how they are implemented. It is well known, for example, that the value 0.1 cannot be represented exactly in the floating point format standardized in IEEE 754 (the most common format, and also the format used in this extension library). In applied science and engineering, this is relatively unimportant because the input values often originate from measurements or approximative computations, which are imprecise to begin with.

This appendix assumes that the reader understands the PAWN language. For more information on PAWN, please read the manual "The PAWN booklet — The Language" which is available from the company homepage.

Implementing the library

The floating point support library consists of the files FLOAT.C and FLOAT.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 FLOAT.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 FLOAT.INC also sets the rational number format for the PAWN compiler to a floating point number (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 floating point extension module together with a fixed point module. Such conflicts can be resolved by removing the `#pragma rational` directive from either module.

The "Implementor'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_FloatInit` and the "clean-up" function is `amx_FloatCleanup`. 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 Implementor's Guide for details.

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

Usage

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

```
#include float
```

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

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

To convert from integers to floating point values, use one of the functions `float` or `strfloat`. The function `float` creates a floating point number with the same integral value as the input value and a fractional part of zero. Function `strfloat` makes a floating 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 floating point format on the left hand. That is, the lines:

```
new a = 10
new Float: b = a
```

are equivalent to:

```
new a = 10
new Float: b = float(a)
```

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

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

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

Due to the limited precision of floating point arithmetic, the calculated value may be *slightly off* the exact/correct answer. Over time, these fractional rounding errors can accumulate. It is therefore advised to avoid comparing two floating point values for bit-for-bit equality. For example, for the novice programmer the following PAWN program may give an unexpected result:

Listing **bad way to compare floating point values (prone to rounding errors)**

```
#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)
    }
```

Instead, you should verify whether the two values lie within a small range —such a comparison range allowing for inexactness in the calculations is typically referred to as ϵ (*epsilon*). The example below makes conveniently use of *chained* relational operators to do the comparison.

Listing **allow minor deflections when comparing floating point values**

```
#include float

const Float: epsilon = 0.00001

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

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

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

For details on floating point inexactness, and improved range checking, see section "Resources".

Native functions

Float:float(value)
: Create a fixed point number with the same (integral) value as the parameter `value`.

Float:strfloat(const string[])
: Create a fixed point number from a string. The string may specify a fractional part, e.g., "`123.45`".

Float:floatmul(Float:oper1, Float:oper2)
: Multiply two fixed point numbers. The user-defined `*` operator forwards to this function.

Float:floatdiv(Float:dividend, Float:divisor)
: Floating point division. The user-defined `/` operator forwards to this function.

Float:floatadd(Float:oper1, Float:oper2)
: Floating point addition. The user-defined `+` operator forwards to this function.

Float:floatsub(Float:oper1, Float:oper1)
: Floating point subtraction. The user-defined `-` operator forwards to this function.

Float:floatfract(Float:value)
: Returns the fractional part of `value`, in floating point format.

floatround(Float:value, floatround_method:method=floatround_round)
: Round a floating point number and return the value as an integer. The rounding method may be one of:

  `floatround_round` round to the nearest integer, where a fractional part of exactly 0.5 rounds upwards (this is the default);

  `floatround_floor` round downwards;

  `floatround_ceil` round upwards;

  `floatround_tozero` round downwards for positive values and upwards for negative values ("truncate");

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

floatcmp(Float:oper1, Float:oper1)

Compares the two operands and returns −1 if `oper1` < `oper2`, +1 if `oper1` > `oper2` and 0 if `oper1` is equal to `oper2`.

Float:floatpower(Float:value, Float:exponent)

Returns `value` raised to the power `exponent`. The exponent may be zero or negative; if the base value is negative, however, the exponent should not have a fractional part.

Float:floatsqroot(Float:value)

Returns the square root of the input number.

Float:floatlog(Float:value, Float:base=10.0)

Returns the logarithm of the input number. The default logarithm base is 10, but you can set a different base.

Float:floatcos(Float:value, anglemode:mode=radian)

Returns the sine of the input value. The (input) angle may be specified in degrees (sexagesimal system, the default), grades (centesimal system) or radian.

Float:floatsin(Float:value, anglemode:mode=radian)

Returns the sine of the input value. The (input) angle may be specified in degrees (sexagesimal system, the default), grades (centesimal system) or radian.

Float:floattan(Float:value, anglemode:mode=radian)

Returns the sine of the input value. The (input) angle may be specified in degrees (sexagesimal system, the default), grades (centesimal system) or radian.

Float:floatabs(Float:value)

Returns the absolute value of the input value.

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.

Float:operator*(Float:oper1, Float:oper2)

Float:operator/(Float:oper1, Float:oper2)

Float:operator+(Float:oper1, Float:oper2)

Float:operator-(Float:oper1, Float:oper2)

Float:operator=(oper)

Float:operator++(Float:oper)

Float:operator−−(Float:oper)

Float:operator−(Float:oper)

Float:operator*(Float:oper1, oper2) (*"*" is commutative*)

Float:operator/(Float:oper1, oper2)

Float:operator/(oper1, Float:oper2)

Float:operator+(Float:oper1, oper2) (*"+" is commutative*)

Float:operator−(Float:oper1, oper2)

Float:operator−(oper1, Float:oper2)

bool:operator>(Float:oper1, Float:oper2)

bool:operator>(Float:oper1, oper2)

bool:operator>(oper1, Float:oper2)

bool:operator>=(Float:oper1, Float:oper2)

bool:operator>=(Float:oper1, oper2)

bool:operator>=(oper1, Float:oper2)

bool:operator<(Float:oper1, Float:oper2)

bool:operator<(Float:oper1, oper2)

bool:operator<(oper1, Float:oper2)

bool:operator<=(Float:oper1, Float:oper2)

bool:operator<=(Float:oper1, oper2)

bool:operator<=(oper1, Float:oper2)

bool:operator==(Float:oper1, Float:oper2)

bool:operator==(Float:oper1, oper2) (*"==" is commutative*)

bool:operator!=(Float:oper1, Float:oper2)

bool:operator!=(Float:oper1, oper2) (*"!=" is commutative*)

bool:operator!(Float:oper)

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**.

Index

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