

# Pawn



embedded scripting language

## Floating Point Support Library

August 2007

### Abstract

---

The “PAWN” programming language has only native support for 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”

The floating point extension module was originally written by Greg Garner, © 1999, Artran, Inc.

INTRODUCTION .....	1
IMPLEMENTING THE LIBRARY .....	2
USAGE .....	3
NATIVE FUNCTIONS .....	5
CUSTOM OPERATORS .....	11
RESOURCES .....	13
INDEX .....	15

“Microsoft” and “Microsoft Windows” are registered trademarks of Microsoft Corporation.

“Linux” is a registered trademark of Linus Torvalds.

“CompuPhase” is a registered trademark of ITB CompuPhase.

Copyright © 2003–2007, ITB CompuPhase; Eerste Industriestraat 19–21, 1401VL  
Bussum, The Netherlands (Pays Bas); telephone: (+31)-(0)35 6939 261  
e-mail: [info@compuphase.com](mailto:info@compuphase.com), WWW: <http://www.compuphase.com>

The information in this manual and the associated software are provided “as is”.  
There are no guarantees, explicit or implied, that the software and the manual  
are accurate.

Requests for corrections and additions to the manual and the software can be  
directed to ITB CompuPhase at the above address.

Typeset with  $\text{\TeX}$  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 an integer (usually 32 or 64 bits wide), 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 “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_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 Implementer’s Guide for details.

The extension module `AMXCONS.C` (console input/output) has some support for floating 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 `float-round` 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  $\epsilon$  (*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

---



---



---

<b>float</b>	Convert integer to floating point
--------------	-----------------------------------

Syntax:	Float: float(value)
---------	---------------------

value	the input value.
-------	------------------

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

See also:	<a href="#">floatround</a> , <a href="#">strfloat</a>
-----------	---

---

<b>floatadd</b>	Add two floating point numbers
-----------------	--------------------------------

Syntax:	Float: floatadd(Float: oper1, Float: oper2)
---------	---

oper1	
-------	--

oper2	The values to add together.
-------	-----------------------------

Returns:	The result: the sum of oper1 and oper2.
----------	---

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

See also:	<a href="#">floatdiv</a> , <a href="#">floatmul</a> , <a href="#">floatsub</a>
-----------	--

---

<b>floatabs</b>	Return the absolute value of a floating point number
-----------------	--

Syntax:	Float: floatabs(Float: value)
---------	-------------------------------

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

Returns:	The absolute value of the parameter.
----------	--------------------------------------

---

<b>floatcmp</b>	Compare two floating point numbers
-----------------	------------------------------------

Syntax:	Float: floatcmp(Float: oper1, Float: oper2)
---------	---

oper1	
-------	--

oper2	The two operands to compare.
-------	------------------------------

-1 if oper1 < oper2, +1 if oper1 > oper2 and 0 if oper1 is equal to oper2.

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

## floatcos

Return the cosine of an angle

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

value	The value to calculate the cosine of.
-------	---------------------------------------

mode	Specifies whether the angle (in parameter <b>value</b> ) is specified in degrees (sexagesimal system), grades (centesimal system) or radian. The default is radian.
------	---

Returns: The result: the cosine of the input number.

See also: `floatsin`, `floattan`

## floatdiv

Divide a floating point number

Syntax:     Float: floatdiv(Float: oper1, Float: oper2)

<code>oper1</code>	The numerator of the quotient.
--------------------	--------------------------------

oper2	The denominator of the quotient.
-------	----------------------------------

Returns:      The result:  $\text{oper1}/\text{oper2}$ .

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

See also: `floatadd`, `floatmul`, `floatsub`

## floatfract

Return the fractional part of a number

Syntax:      Float: floatfract(Float: value)

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

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

See also: [floatround](#)



---

**floatlog** Return the logarithm of a value

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

          value         The value to calculate the logarithm of.

          base         The logarithmic base to use; the default base is 10.

Returns:     The result: the logarithm of the input number.

Notes:       This function raises a “domain” error if the input value is zero or negative.

See also:     [floatpower](#)

---

**floatmul** Multiply two floating point numbers

Syntax:     Float: floatmul(Float: oper1, Float: oper2)

          oper1

          oper2         The two operands to multiply.

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

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

See also:     [floatadd](#), [floatdiv](#), [floatsub](#)

---

**floatpower** Raise a floating point number to a power

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

          value         The value to raise to a power; this is a floating point number.

          exponent     The exponent is also a floating pointer number. The exponent may be zero or negative.

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

See also:     [floatlog](#), [floatsqroot](#)

---

**floatround** Round a floating point number to an integer value

Syntax:     `floatround(Float: value,`  
                    `floatround_method: method=floatround_round)`

**value**       The value to round.

**method**     The rounding method may be one of:

`floatround_round`  
                        round to the nearest integer; 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 up-  
                        wards for negative values (“truncate”);

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:    [floatfract](#)

---

**floatsin** Return the sine of an angle

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

**value**       The value to calculate the sine of.

**mode**       Specifies whether the angle (in parameter **value**) is  
                        specified in degrees (sexagesimal system), grades (cen-  
                        tesimal system) or radian. The default is radian.

Returns:     The result: the sine of the input number.

See also:    [floatcos](#), [floattan](#)

---

<b>floatsgroot</b>	Return the square root of a value
--------------------	-----------------------------------

Syntax:	Float: floatsgroot(Float: 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:	<a href="#">floatpower</a>

---

<b>floatsub</b>	Subtract a floating point number from another
-----------------	---

Syntax:	Float: floatsub(Float: oper1, Float: oper2)
	oper1
	oper2      The values to add together.
Returns:	The result: the oper1 minus oper2.
Notes:	The user-defined + operator forwards to this function.
See also:	<a href="#">floatdiv</a> , <a href="#">floatmul</a> , <a href="#">floatsub</a>

---

<b>floattan</b>	Return the tangent of an angle
-----------------	--------------------------------

Syntax:	Float: floattan(Float: value, anglemode: mode=radian)
	value      The value to calculate the tangent of.
	mode      Specifies whether the angle (in parameter value) is specified in degrees (sexagesimal system), grades (centesimal system) or radian. The default is radian.
Returns:	The result: the tangent of the input number.
See also:	<a href="#">floatcos</a> , <a href="#">floatsin</a>

---

<b>strfloat</b>	Convert from text (string) to floating point
-----------------	--

Syntax:	Float: strfloat(const string[])
---------	---------------------------------

**string**      The string containing a floating point number in characters. This may be either a packed or unpacked string. The string may specify 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.

**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](http://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](http://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**.

<b>!</b> <hr/> <b>#include</b> , 2 <b>#pragma rational</b> , 2, 3	Floating point, 1, 13 floatlog, 7 floatmul, 7 floatpower, 7 floatround, 3, 8 floatsin, 8 floatsqroot, 9 floatsub, 9 floattan, 9 Forbidden operators, 3
<b>A</b> <hr/> Absolute value, 5 Abstract Machine, 2 Adobe Acrobat, 13	
<b>B</b> <hr/> Base 10, <i>See</i> Decimal arithmetic Base 2, <i>See</i> Binary arithmetic <i>Bush, B.M.</i> , 13	
<b>C</b> <hr/> cell, 1 Centesimal system, 6, 8, 9 Chained relational operators, 4 Console module, 2 Cosine, 6	<b>H</b> <hr/> Host application, 2
<b>D</b> <hr/> DLL, 2	<b>I</b> <hr/> IEEE 754, 1, 13
<b>E</b> <hr/> Exponentiation, 7	<b>L</b> <hr/> Linux, 2 Literal numbers, 3 Logarithm, 7
<b>F</b> <hr/> Fixed point module, 2 float, 3, 5 floatabs, 5 floatadd, 5 floatcmp, 5 floatcos, 6 floatdiv, 6 floatfract, 3, 6	<b>M</b> <hr/> Microsoft Windows, 2 Modulus, 3
	<b>N</b> <hr/> Native functions, 2 registering, 2
	<b>O</b> <hr/> Operators forbidden, 3 user-defined, 2, 3, 11
	<b>P</b> <hr/> Prefix file, 2 Preprocessor directive, 2

R		Sine, 8
	Radian, 6, 8, 9	Square root, 9
	Registering, 2	strfloat, 3, 9
S		
	Sexagesimal system, 6, 8, 9	Tangent, 9
	Shared library, 2	
	Significant digits, 1	