Micromega Corporation

Using the uM-FPU with the Javelin Stamp

Introduction

The uM-FPU is a 32-bit floating point coprocessor that can be easily interfaced with the Javelin Stamp to provide support for 32-bit IEEE 754 floating point operations and long integer operations. The uM-FPU is easy to connect, and requires only two pins on the Javelin Stamp. The only external component required for operation is a protection resistor on the bidirectional data line.

uM-FPU Features

- 8-pin integrated circuit.
- Bi-directional serial interface requires only two wires for connection.
- Sixteen 32-bit general purpose registers for storing floating point or long integer values
- Five 32-bit temporary registers with support for nested calculations (i.e. parenthesis)
- Floating Point Operations
  - Set, Add, Subtract, Multiply, Divide
  - Sqrt, Log, Log10, Exp, Exp10, Power, Root
  - Sin, Cos, Tan
  - Asin, Acos, Atan, Atan2
  - Floor, Ceil, Round, Min, Max, Fraction
  - Negate, Abs, Inverse
  - Convert Radians to Degrees
  - Convert Degrees to Radians
  - Compare, Status
- Long Integer Operations
  - Set, Add, Subtract, Multiply, Divide, Unsigned Divide
  - Negate, Abs
  - Compare, Unsigned Compare, Status
- Conversion Functions
  - Convert 8-bit and 16-bit integers to floating point
  - Convert 8-bit and 16-bit integers to long integer
  - Convert long integer to floating point
  - Convert floating point to long integer
  - Convert floating point to ASCII
  - Convert floating point to formatted ASCII
  - Convert long integer to ASCII
  - Convert long integer to formatted ASCII
  - Convert ASCII to floating point
  - Convert ASCII to long integer

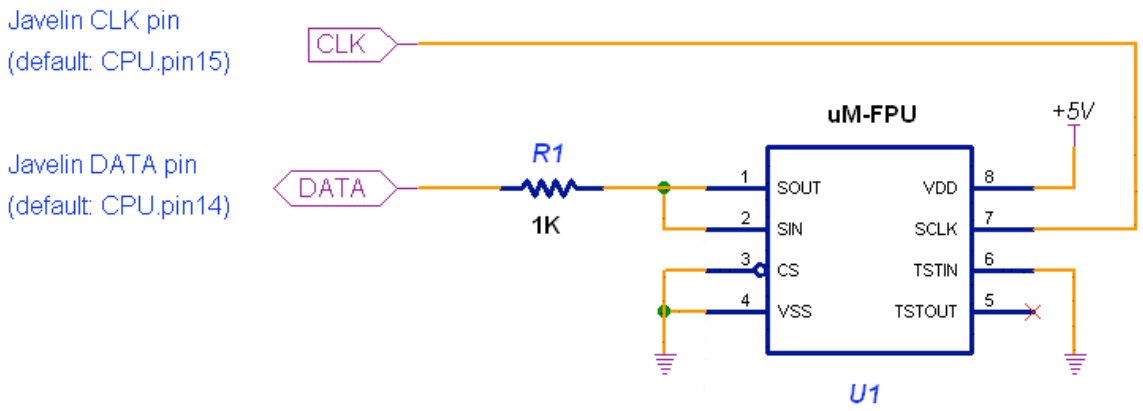
- uM-FPU Javelin Stamp package provides a full set of support routines for easy implementation.

Connecting the uM-FPU to the Javelin Stamp

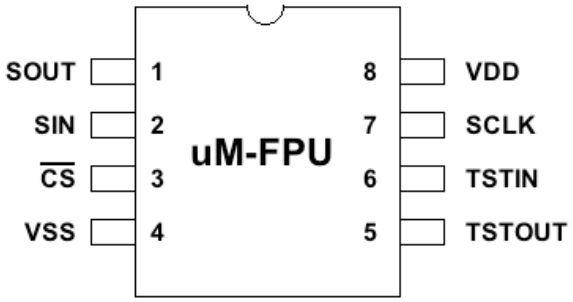
The uM-FPU requires just two pins for interfacing to the Javelin Stamp. The communication is implemented using a bidirectional serial interface that requires a clock pin and a data pin. The default setting for these pins are:

```
final static int CLOCK_PIN = CPU.pin15;
final static int DATA_PIN  = CPU.pin14;
```

The settings for these pins can be changed to suit your application. The support routines assume that the uM-FPU chip is always selected, so `CLOCK_PIN` and `DATA_PIN` should not be used for other connections as this will likely result in loss of synchronization between the Javelin Stamp and the uM-FPU coprocessor.



uM-FPU Pin Assignment



PIN DESCRIPTION

SOUT	SPI Output
SIN	SPI Input
$\overline{CS}$	Chip Select
VSS	Ground
TSTOUT	Test Output
TSTIN	Test Input
SCLK	SPI Clock
VDD	Power Supply Voltage (+5V)

Using the uM-FPU Javelin Stamp Package

A package is provided to handle all of the communication between the Javelin Stamp and the uM-FPU. The package is located in the ~\lib\com\micromegacorp\math\hardware and contains the following classes:

Name	Description
Float32	32-bit IEEE 754 floating point
Float32Math	Additional floating point math functions
Float32Constant	Floating point constants
Int32	32-bit long integer
FpuData	32-bit basic data type (extended by Float32 and Int32)
Fpu	Interface for the uM-FPU floating point coprocessor

The Fpu and FpuData classes provide the low-level support for the uM-FPU and except for the Fpu.reset and Fpu.version methods, are not called directly by the user. The Float32, Float32Math, and FloatConstant classes provide support for 32-bit floating point values. The Int32 class provides support for 32-bit integer values. The API documentation for each of these classes is available in Javadoc HTML format.

The following statement should be added to any routine that uses the uM-FPU math package.

```
package com.micromegacorp.math.hardware;
```

In order to ensure that the JavelinStamp and the uM-FPU coprocessor are synchronized, a reset call must be done at the start of every program. The Fpu.reset method resets the uM-FPU, confirms communications, and returns `true` if successful, or `false` if the reset fails. An example of a typical reset is as follows:

```
if (!Fpu.reset()) {
  System.out.println("uM-FPU not detected.");
  return;
}
```

The version number of the support software and uM-FPU chip can be displayed with the following statement:

```
System.out.println(Fpu.version());
```

The uM-FPU contains sixteen 32-bit registers which are used to store floating point or long integer values, but the Javelin math package uses a caching scheme to provide support for as many floating point or long integer variables as the Javelin memory will accommodate.

Using Float32 Objects

Creating Float32 Objects

There are four constructors for Float32 objects, which you can use to:

1) Create a new Float32 initialized to zero.

```
Float32 fnum1 = new Float32();
```

2) Create a new Float32 initialized to the value of another Float32.

```
Float32 fnum2 = new Float32(fnum1);
```

3) Create a new Float32 initialized to the 32-bit floating point value specified by two integers.

```
// set initial value to 10000000.0 (hex: 0x4B189680)
Float32 fnum3 = new Float32((short)0x4B18, (short)0x9680);
```

4) Create a new Float32 initialized to a 16-bit integer value (sign extended).

```
// set initial value to value of integer n
Float32 fnum3 = new Float32(n);

// set initial value to 250
Float32 fnum4 = new Float32(250);
```

A handy utility program called `uM-FPU Converter` is available to convert between 32-bit floating point values and hexadecimal values.

Setting the value of Float32 Objects

There are five methods which set the value of a Float32 object. They can be used to:

1) Set Float32 to the value of another Float32.

```
// set fnum2 to value of fnum1
fnum2.set(fnum1);
```

2) Set Float32 to the 32-bit floating point value specified by two integers.

```
// set fnum3 to 10000000.0 (hex: 0x4B189680)
fnum3.set((short)0x4B18, (short)0x9680);
```

The (short) cast is used since the default data type for a hex constant is signed integer and the compiler will generate an error if a positive number is greater than 32767 (e.g. 0x8000 and above). Using the (short) allows 16-bit unsigned values to be specified.

3) Set Float32 to a 16-bit integer value (sign extended).

```
// set fnum3 to value of integer n
fnum3.set(n);

// set fnum4 to 250
fnum4.set(250);
```

4) Set Float32 to the converted value of a String.

```
// set fnum4 to 625.0125
fnum4.set("625.0125");
```

The string conversion will ignore leading spaces and stop at the first character that is not part of a valid floating point number. Both normal and exponential formats are supported.

5) Set Float32 to the converted value of a StringBuffer.

```
// set fnum5 to  -0.00000015
StringBuffer sbuf = new StringBuffer("-1.5e-7");
fnum5.set(sbuf);
```

Setting a Float32 to the value of a 16-bit integer, a String or a StringBuffer requires the uM-FPU to perform the operation. Since the uM-FPU must be reset before functions are used, care must be taken that the method of declaring a variable doesn't result in Java executing one of these constructors before the uM-FPU has been initialized. This can occur when a variable is defined as a static class variable in the main class. To avoid this, restrict the use of the 16-bit integer, String and StringBuffer constructors to creating objects inside the code after the Fpu.reset method has been called. If a Float32 object is used before a reset is done, the value will be set to Not-a-Number (NaN).

Float32 Operators

The Float32 class provides methods for each of the following floating point math operators: add, subtract, multiply, divide, compare, abs, negate and inverse. The abs, negate and inverse unary operators have no additional arguments.

```
// value = -value
value.negate();

// value = |value|
value.negate();

// value = 1 / value
value.inverse();
```

The other operators take an argument that specifies the value to use with the operation. There are three methods supported for each of the operators. The following examples show the various methods of the add operator.

1) Use the value of another Float32.

```
// add fnum2 to value of fnum1
fnum2.add(fnum1);
```

2) Use the 32-bit floating point value specified by two integers.

```
// add fnum3 to 10000000.0 (hex: 0x4B189680)
fnum3.add((short)0x4B18, (short)0x9680);
```

The (short) cast is used since the default data type for a hex constant is signed integer and the javelin compiler will generate an error if a positive number is greater than 32767 (e.g. 0x8000 and above). Using the (short) cast allows 16-bit unsigned values to be specified.

3) Use a 16-bit integer value (sign extended).

```
// add the value of integer n to fnum3
fnum3.add(n);
```

```
// add 250 to fnum4
fnum4.add(250);
```

Examples showing other operators are as follows:

```
// fnum1 = fnum1 - fnum2
fnum1.subtract(fnum2);

// fnum1 = fnum1 * fnum2
fnum1.multiply(fnum2);

// fnum1 = fnum1 / fnum2
fnum1.divide(fnum2);

// fnum1 = fnum1 / 100
fnum1.divide(100);
```

The intValue method is used to convert a floating point value to an integer value.

```
// get the integer value of angle
n =angle.intValue();
```

Comparing Float32 Values

The compare operator returns an integer that is greater than zero if the Float32 is greater then the value passed, equal to zero if the values are the same, or less than zero if the Float32 is less than the value.

```
// Compare fnum1 and fnum2
if (fnum1.compare(fnum2) > 0) System.out.println("fnum1 > fnum2");
if (fnum1.compare(fnum2) == 0) System.out.println("fnum1 == fnum2");
if (fnum1.compare(fnum2) < 0) System.out.println("fnum1 < fnum2");

// Check if fnum1 >= 1000
if (fnum1.compare(1000) >= 0) System.out.println("fnum1 >= 1000");
```

The equals operator returns a Boolean that is `true` if the Float32 is equal to the value passed, and `false` otherwise.

```
// Check if fnum1 == fnum2
if (fnum1.equals(fnum2)) System.out.println("fnum1 == fnum2");
```

The isZero, isNegative, isInfinity, and isNaN methods are used to check the status of a Float32 object.

```
// Check if fnum1 is negative.
if (fnum1.isNegative()) System.out.println("fnum1 is negative");

// Check if fnum2 is zero (checks for both +0 and -0)
if (fnum2.isZero()) System.out.println("fnum2 is zero");

// Check if fnum1 is infinite.
if (fnum1.isInfinite()) System.out.println("fnum1 is infinite");

// Check if fnum1 is Not a Number (NaN).
if (fnum1.isNaN()) System.out.println("fnum3 is not a number");
```

Printing Float32 Values

The toString method converts the value of a Float32 to a String. The floating point value will be displayed with up to eight digits of precision. Numbers that are very large or very small are displayed in exponential format. The toHexString method displays the the 32-bit binary floating point value as a hexadecimal number.

```
// print the value of an fnum4 in floating point and hex
fnum4.set(100);
System.out.print("fnum4 = ");
System.out.print(fnum4.toString());
System.out.print(", ");
System.out.println(fnum4.toHexString());
```

The printed result would be: `fnum4 = 100.0, 0x42C80000`

Float32Constant Class

The Float32Constant class provides methods to return the value of the following floating point constants: minimum Float32 value, maximum Float32 value, positive infinity, negative infinfity, NaN, pi and e.

```
// Print the maximum Float32 value
fnum1.set(Float32Constant.maxValue());
System.out.println(fnum1.toString());

// Print the value of pi
fnum1.set(Float32Constant.pi());
System.out.println(fnum1.toString());
```

Float32Math Class

The Float32Math class provides a wide variety of math functions. They are as follows: abs, fraction, min, max, sin, cos, tan, asin, acos, atan, atan2, pow, root, sqrt, exp, exp10, log, log10, floor, ceil, round, toDegrees and toRadians. All of the methods take one or two Float32 objects as arguments and return the result as a reference to a static Float32 object. Normally the return value is saved to another variable for use laster in the program. Some examples are as follows:

```
// set fnum2 to the square root of fnum1
Fnum2.set(Float32Math.sqrt(fnum1));

// set fnum1 to the sine of an angle
fnum1.set(Float32Math.sin(angle));

// set fnum3 = fnum1 ** fnum2
fnum3.set(Float32Math.pow(fnum1, fnum2));
```

High and Low Methods

The high and low methods are used to get the high 16 bits and low 16 bits of the 32-bit binary representation of the floating point value. In most cases the user does not need to access these values directly.

```
// get the high 16 bits of fnum1
high = fnum1.high();

// get the low 16 bits of fnum1
low = fnum1.low();
```

Using Int32 Objects

Creating Int32 Objects

There are four constructors for Int32 objects, which can be used to:

1) Create a new Int32 initialized to zero.

```
Int32 num1 = new Int32();
```

2) Create a new Int32 initialized to the value of another Int32.

```
Int32 num2 = new Int32(fnum1);
```

3) Create a new Int32 initialized to the 32-bit long integer value specified by two integers.

```
// set initial value to 1000000 (hex: 0x000F4240)
Int32 num3 = new Int32((short)0x000F, (short)0x4240);
```

4) Create a new Int32 initialized to a 16-bit integer value (sign extended).

```
// set initial value to value of integer n
Int32 num3 = new Int32 (n);

// set initial value to 250
Int32 num4 = new Int32(250);
```

Setting the value of Int32 Objects

There are five methods which set the value of an Int32 object. They are as follows:

1) Set Int32 to the value of another Int32.

```
// set fum2 to value of num1
num2.set(num1);
```

2) Set Int32 to the 32-bit long integer value specified by two integers.

```
// set num3 to 10000000.0 (hex: 0x4B189680)
num3.set((short)0x4B18, (short)0x9680);
```

The (short) cast is used since the default data type for a hex constant is signed integer and the compiler will generate an error if a positive number is greater than 32767 (e.g. 0x8000 and above). Using the (short) allows 16-bit unsigned values to be specified.

3) Set Int32 to a 16-bit integer value (sign extended).

```
// set num3 to value of integer n
num3.set(n);

// set num4 to 250
num4.set(250);
```

4) Set Int32 to the converted value of a String.

```
// set num4 to 99000
num4.set("99000");
```

The string conversion will ignore leading spaces and stop at the first character that is not part of a valid integer number.

5) Set Int32 to the converted value of a StringBuffer.

```
// set num5 to 1000000
StringBuffer sbuf = new StringBuffer("1000000");
num5.set(sbuf);
```

Setting an Int32 to the value of a 16-bit integer, a String or a StringBuffer requires the uM-FPU to perform the operation. Since the uM-FPU must be reset before functions are used, care must be taken that the method of declaring a variable doesn't result in Java executing one of these constructors before the uM-FPU has been initialized. This can occur when a variable is defined as a static class variable in the main class. To avoid this, restrict the use of the 16-bit integer, String and StringBuffer constructors to creating objects inside the code after the Fpu.reset method has been called. If an Int32 is used before a reset is done the value will be set to 0x7C80000 (this is the floating point NaN value).

Int32 Operators

The Int32 class provides methods for each of the following integer math operators: add, subtract, multiply, divide, unsigned divide, compare, unsigned compare, abs and negate. The abs, negate and inverse unary operators have no additional arguments.

```
// value = -value
value.negate();

// value = |value|
value.negate();
```

The other operators take an argument that specifies the value to use with the operation. There are three methods supported for each of the operators. The following examples show the various methods of the add operator.

1) Use the value of another Int32.

```
// add num2 to value of num1
num2.add(num1);
```

2) Use the 32-bit long integer value specified by two integers.

```
// add num3 to 10000000.0 (hex: 0x4B189680)
num3.add((short)0x4B18, (short)0x9680);
```

The (short) cast is used since the default data type for a hex constant is signed integer and the javelin compiler will generate an error if a positive number is greater than 32767 (e.g. 0x8000 and above). Using the (short) cast allows 16-bit unsigned values to be specified.

3) Use a 16-bit integer value (sign extended).

```
// add the value of integer n to num3
num3.add(n);

// add 250 to num4
num4.add(250);
```

Examples showing other operators are as follows:

```
// num1 = num1 - num2
num1.subtract(num2);

// num1 = num2 * num1
num1.multiply(num2);

// num1 = num1 / num2
fnum1.divide(fnum2);

// num1 = num1 / 100
num1.divide(100);
```

Comparing Int32 Values

The compare operator returns an integer that is greater than zero if the Int32 is greater then the value passed, equal to zero if the values are the same, or less than zero if the Int32 is less than the value.

```
// Compare num1 and num2
if (num1.compare(num2) > 0) System.out.println("num1 > num2");
if (num1.compare(num2) == 0) System.out.println("num1 == num2");
if (num1.compare(num2) < 0) System.out.println("num1 < num2");

// Check if num1 >= 1000
if (num1.compare(1000) >= 0) System.out.println("num1 >= 1000");
```

The equals operator returns a Boolean that is `true` if the Int32 is equal to the value passed, and `false` otherwise.

```
// Check if num1 == num2
if (num1.equals(num2)) System.out.println("num1 == num2");
```

The isZero and isNegative methods are used to check the status of an Int32 object .

```
// Check if num1 is negative.
if (num1.isNegative()) System.out.println("num1 is negative");

// Check if num2 is zero
if (num2.isZero()) System.out.println("num2 is zero");
```

Printing Int32 Values

The toString method converts the signed value of an Int32 to a String, and the utoString method converts the unsigned value of an Int32 to a String. The toHexString method displays the 32-bit value in hexadecimal format.

```
// print the value of an num4 in 32-bit integer and hex format
fnum.set(100);
System.out.print("num4 = ");
System.out.print(num4.toString());
System.out.print(", ");
System.out.println(num4.toHexString());
```

The printed result would be: `num4 = 100, 0x00000064`

High and Low Methods

The high and low methods are used to get the high 16 bits and low 16 bits of the 32-bit binary representation of 32-bit integer value. In most cases the user does not need to access these values directly.

```
// get the high 16 bits of num1
high = num1.high();

// get the low 16 bits of num1
low = num1.low();
```

Sample Code

```
// The following example takes an integer value representing the
// diameter of a circle in centimeters, converts the value to inches,
// and calculates the circumference in inches and the area in square
// inches.

System.out.println("\r\nConversion Example");
System.out.println("------------------");

// create the Float32 constants
Float32 pi    = new Float32(Float32Constant.pi());          // pi
Float32 f2_54 = new Float32((short)0x4022, (short)0x8F5C);  // 2.54

// create the Float32 variables
Float32 diameterIn    = new Float32();
Float32 circumference = new Float32();
Float32 area          = new Float32();

int diameter = 25;
System.out.print("Diameter (cm):       ");
System.out.println(diameter);

// convert diameter from centimeters to inches
// diameterIn = diameter / 2.54
diameterIn.set(diameter);
diameterIn.divide(f2_54);
System.out.print("Diameter (in.):      ");
System.out.println(diameterIn.toString());

// circumference = diameter * pi;
circumference.set(diameterIn);
circumference.multiply(pi);
System.out.print("Circumference (in.): ");
System.out.println(circumference.toString());

// area = (diameter / 2)^2 * pi;
area.set(diameterIn);
area.divide(2);
area.multiply(area);
area.multiply(pi);
System.out.print("Area (sq.in.):       ");
System.out.println(area.toString());

System.out.println("\r\nDone.");
```

Appendix A
Reference for uM-FPU Javelin Stamp methods

Fpu Methods

reset	Reset the uM-FPU and confirm communications.
version	Get uM-FPU version string.

Float32 Methods

set	Set this Float32 value.
add	Add another value to this Float32.
subtract	Subtract a value from this Float32.
multiply	Multiply this Float32 by another value.
divide	Divide this Float32 by another value.
compare	Compare this Float32 with another value.
abs	Set this Float32 to its absolute value.
negate	Negate this Float32.
inverse	Set this Float32 value to 1/value.
equals	Check if this Float32 is equal to another Float32.
isZero	Check if this Float32 is zero.
isNegative	Check if this Float32 is negative.
isInfinite	Check if this Float32 is an infinity.
isNaN	Check if this Float32 is Not-a-Number (NaN).
intValue	Get the integer value of this Float32.
toString	Convert the Float32 value to a String.
toHexString	Convert the Float32 value to a hexadecimal String.
high	Get the high 16 bits of the Float32 value.
low	Get the low 16 bits of the Float32 value.

Float32Math Methods

abs	Return the absolute value of a Float32.
fraction	Return the fractionall portion of a Float32.
min	Return the minimum Float32.
max	Return the minimum Float32.
sin	Return the sine of an angle.
cos	Return the cosine of an angle.
tan	Return the tangent of an angle.
asin	Return the arc sine of an angle.
acos	Return the arc cosine of an angle.
atan	Return the arc tangent of an angle.
atan2	Return angle for coordinates (x, y).
pow	returns a value raised to the power of another value.
root	returns the nth root of a value
sqrt	returns the square root of a value
exp	Return e raised to the value.
exp10	Return 10 raised to value.
log	Return the log base e of the value.
log10	Return the log base e of the value.
floor	Return the largest integer less then the value.
ceil	Return the smallest integer greater then the value.
round	Return the largest integer less then the value.
toDegrees	Returns the value of radians converted to degrees.
toRadians	returns the value of degrees converted to radians.

Float32Constant Methods

minValue	Returns the minimum Float32 value.
maxValue	Returns the maximum Float32 value.
positiveInfinity	Returns the value of positive infinity.
negativeInfinity	Returns the value of negative infinity.
nan	Returns the value of Not-a-Number (NaN).
pi	Returns the value of pi.
e	Returns the value of e.

Int32 Methods

set	Set this Int32 value.
add	Add another value to this Int32.
subtract	Subtract a value from this Int32.
multiply	Multiply this Int32 by another value.
divide	Divide this Int32 by another value.
udivide	Unsigned divide of this Int32 by another value.
remainder	Sets this Int32 to the remainder of the last Int32 divide or udivide.
compare	Compare this Int32 with another value.
ucompare	Unsigned compare of this Int32 with another value.
abs	Set this Int32 to its absolute value.
negate	negate this Int32.
equals	check if this Int32 is equal to another Int32.
isZero	check if this Int32 is zero.
isNegative	check if this Int32 is negative.
toString	convert the Int32 value to a String.
toHexString	convert the Int32 value to a hexadecimal String.
high	get the high 16 bits of the Int32 value.
low	get the low 16 bits of the Int32 value.

The API documentation for each of these classes is available in Javadoc HTML format. Please refer to these documents for more details.

Appendix B
Floating Point Numbers

Floating point numbers can store both very large and very small values by "floating" the window of precision to fit the scale of the number. Fixed point numbers can't handle very large or very small numbers and are prone to loss of precision when numbers are divided. The representation of floating point numbers used by the uM-FPU is defined by the IEEE 754 standard.

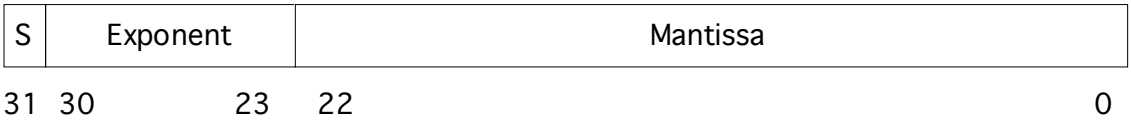
The range of numbers that can be handled by the uM-FPU is approximately $\pm 10^{38.53}$.

.

IEEE 754 32-bit Floating Point Representation

IEEE floating point numbers have three components: the sign, the exponent, and the mantissa. The sign indicates whether the number is positive or negative. The exponent has an implied base of two. The mantissa is composed of the fraction.

The 32-bit IEEE 754 representation is as follows:



Sign Bit (S)
The sign bit is 0 for a positive number and 1 for a negative number.

Exponent
The exponent field is an 8-bit field that stores the value of the exponent with a bias of 127 that allows it to represent both positive and negative exponents. For example, if the exponent field is 128, it represents an exponent of one (128 – 127 = 1). An exponent field of all zeroes is used for denormalized numbers and an exponent field of all ones is used for the special numbers +infinity, -infinity and Not-a-Number (described below).

Mantissa
The mantissa is a 23-bit field that stores the precision bits of the number. For normalized numbers there is an implied leading bit equal to one.

Special Values

Zero

A zero value is represented by an exponent of zero and a mantissa of zero. Note that +0 and –0 are distinct values although they compare as equal.

Denormalized

If an exponent is all zeros, but the mantissa is non-zero the value is a denormalized number. Denormalized numbers are used to represent very small numbers and provide for an extended range and a graceful transition towards zero on underflows. Note: The uM-FPU does not support operations using denormalized numbers.

Infinity

The values +infinity and –infinity are denoted with an exponent of all ones and a fraction of all zeroes. The sign bit distinguishes between +infinity and –infinity. This allows operations to continue past an overflow. A nonzero number divided by zero will result in an infinity value.

Not A Number (NaN)

The value NaN is used to represent a value that does not represent a real number. An operation such as zero divided by zero will result in a value of NaN. The NaN value will flow through any mathematical operation. Note: The uM-FPU initializes all of its registers to NaN at reset, therefore any operation that uses a register that has not been previously set with a value will produce a result of NaN.

Some examples of IEEE 754 32-bit floating point values displayed as Javelin Stamp hex constants are as follows:

```
(short)0x0000, (short)0x0000     // 0.0
(short)0x3DCC, (short)0xCCCD     // 0.1
(short)0x3F00, (short)0x0000     // 0.5
(short)0x3F40, (short)0x0000     // 0.75
(short)0x3F7F, (short)0xF972     // 0.9999
(short)0x3F80, (short)0x0000     // 1.0
(short)0x4000, (short)0x0000     // 2.0
(short)0x402D, (short)0xF854     // 2.7182818 (e)
(short)0x4049, (short)0x0FDB     // 3.1415927 (pi)
(short)0x4120, (short)0x0000     // 10.0
(short)0x42C8, (short)0x0000     // 100.0
(short)0x447A, (short)0x0000     // 1000.0
(short)0x449A, (short)0x522B     // 1234.5678
(short)0x4974, (short)0x2400     // 1000000.0
(short)0x8000, (short)0x0000     // -0.0
(short)0xBF80, (short)0x0000     // -1.0
(short)0xC120, (short)0x0000     // -10.0
(short)0xC2C8, (short)0x0000     // -100.0
(short)0x7FC0, (short)0x0000     // NaN (Not-a-Number)
(short)0x7F80, (short)0x0000     // +inf
(short)0xFF80, (short)0x0000     // -inf
```