Using uM-FPU V2 with the Javelin Stamp™

Micromega Corporation

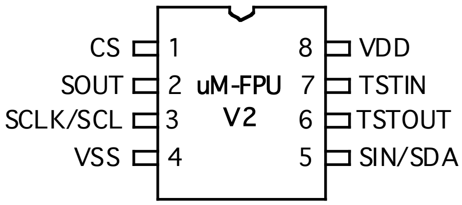
Introduction

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 32-bit long integer operations. The uM-FPU supports both I^2C and 2-Wire SPI connections.

uM-FPU V2 Features

- 8-pin integrated circuit.
- I^2C compatible interface up to 400 kHz
- SPI compatible interface up to 4 Mhz
- 32 byte instruction buffer
- 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. parentheses)
- 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
  - Read, Compare, Status
- Long Integer Operations
  - Set, Add, Subtract, Multiply, Divide, Unsigned Divide
  - Increment, Decrement, Negate, Abs
  - And, Or, Xor, Not, Shift
  - Read 8-bit, 16-bit, and 32-bit
  - 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 formatted ASCII
  - Convert long integer to formatted ASCII
  - Convert ASCII to floating point
  - Convert ASCII to long integer
- User Defined Functions can be stored in Flash memory
  - Conditional execution
  - Table lookup
  - N^{th} order polynomials

Pin Diagram and Pin Description



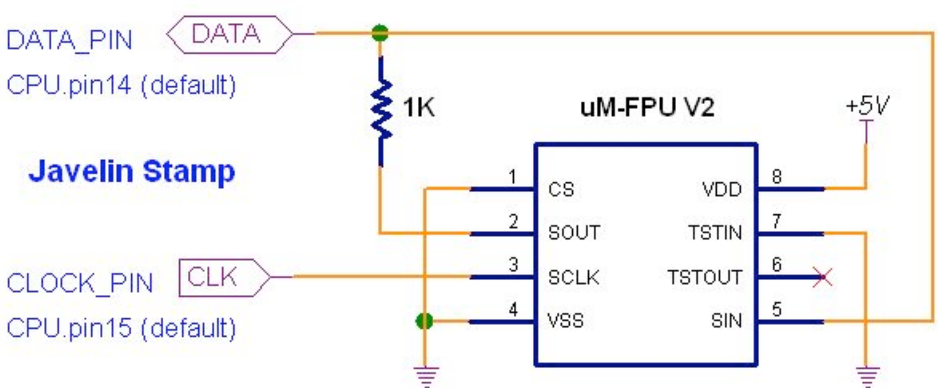
Pin	Name	Type	Description
1	CS	Input	Chip Select
2	SOUT	Output	SPI Output Busy/Ready
3	SCLK SCL	Input	SPI Clock I^2C Clock
4	VSS	Power	Ground
5	SIN SDA	Input In/Out	SPI Input I^2C Data
6	TSTOUT	Output	Test Output
7	TSTIN	Input	Test Input
8	VDD	Power	Supply Voltage

Connecting uM-FPU V2 to the Javelin Stamp using 2-wire SPI

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

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.

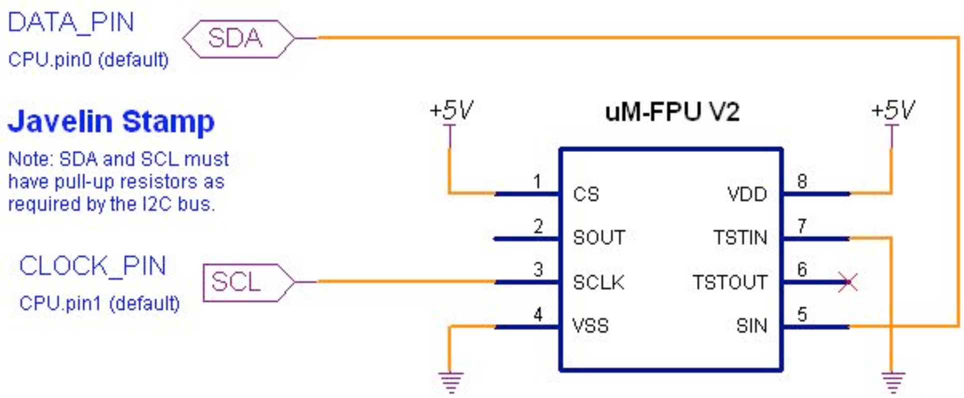


Connecting uM-FPU V2 to the Javelin Stamp using I^2C

The uM-FPU V2 can also be connected using an I^2C interface. The default slaveID for the uM-FPU is $C8. The default settings for the I^2C pins is:

```
final static int DATA_PIN  = CPU.pin0;
final static int CLOCK_PIN = CPU.pin1;
```

The settings for these pins can be changed to suit your application.



An Introduction to the uM-FPU

The following section provides an introduction to the uM-FPU using Javelin methods for all of the examples. For more detailed information about the uM-FPU, please refer to the following documents:

uM-FPU V2 Datasheet	functional description and hardware specifications
uM-FPU V2 Instruction Set	full description of each instruction

uM-FPU Registers

The uM-FPU contains sixteen 32-bit registers, numbered 0 through 15, which are used to store floating point or long integer values. Register 0 is reserved for use as a temporary register and is modified by some of the uM-FPU operations. Registers 1 through 15 are available for general use. Arithmetic operations are defined in terms of an A register and a B register. Any of the 16 registers can be selected as the A or B register.

uM-FPU Registers

	0	32-bit Register
	1	32-bit Register
A →	2	32-bit Register
	3	32-bit Register
	4	32-bit Register
B →	5	32-bit Register
	6	32-bit Register
	7	32-bit Register
	8	32-bit Register
	9	32-bit Register
	10	32-bit Register
	11	32-bit Register
	12	32-bit Register
	13	32-bit Register
	14	32-bit Register
	15	32-bit Register

The `FADD` instruction adds two floating point values and is defined as A = A + B. To add the value in register 5 to the value in register 2, you would do the following:

- Select register 2 as the A register
- Select register 5 as the B register
- Send the `FADD` instruction (A = A + B)

We'll look at how to send these instructions to the uM-FPU in the next section.

Register 0 is a temporary register. If you want to use a value later in your program, store it in one of the registers 1 to 15. Several instructions load register 0 with a temporary value, and then select register 0 as the B register. As you will see shortly, this is very convenient because other instructions can use the value in register 0 immediately.

Sending Instructions to the uM-FPU

Appendix A contains a table that gives a summary of each uM-FPU instruction, and enough information to follow the examples in this document. For a detailed description of each instruction, refer to the document entitled *uM-FPU Instruction Set*.

To send instructions to the uM-FPU the `Fpu.startWrite`, `Fpu.write`, and `Fpu.stop` methods are used as follows:

```
Fpu.startWrite();
Fpu.write(Fpu.FADD+5);
```

```
Fpu.stop();
```

The `Fpu.startWrite` and `Fpu.stop` methods are used to indicate the start and end of a write transfer. A write transfer will often consist of several instructions and data. Up to 32 bytes can be sent in a single write transfer. If more then 32 bytes are required, the `Fpu.wait` method must be called to wait for the uM-FPU to be ready before starting another write transfer and sending more instructions and data.

The `Fpu.write` method can have up to four parameters. Each parameter is an 8-bit value that represents an instruction or data to be sent to the uM-FPU. All instructions start with an opcode that tells the uM-FPU which operation to perform. The `Fpu` class contains definitions for all of the uM-FPU V2 opcodes. Some instructions require additional data or arguments, and some instructions return data. The most common instructions (the ones shown in the first half of the table in Appendix A), require a single byte for the opcode. For example:

```
Fpu.write(Fpu.SQRT);
```

The instructions in the last half of the table, are extended opcodes, and require a two byte opcode. The first byte of extended opcodes is defined as `XOP`. To use an extended opcode, you send the `XOP` byte first, followed by the extended opcode. For example:

```
Fpu.write(Fpu.XOP, Fpu.ATAN);
```

Some of the most commonly used instructions use the lower 4 bits of the opcode to select a register. This allows them to select a register and perform an operation at the same time. Opcodes that include a register value are defined with the register value equal to 0, so using the opcode by itself selects register 0. The following command selects register 0 as the B register then calculates A = A + B.

```
Fpu.write(Fpu.FADD);
```

To select a different register, you simply add the register value to the opcode. The following command selects register 5 as the B register then calculates A = A + B.

```
Fpu.write(Fpu.FADD+5);
```

Let's look at a more complete example. Earlier, we described the steps required to add the value in register 5 to the value in register 2. The command to perform that operation is as follows:

```
Fpu.write(Fpu.SELECTA+2, Fpu.FADD+5);
```

Description:

`SELECTA+2`	select register 2 as the A register
`FADD+5`	select register 5 as the B register and calculate A = A + B

It's a good idea to use constant definitions to provide meaningful names for the registers. This makes your program code easier to read and understand. The same example using constant definitions would be:

```
final static int Total = 2    // total amount  (uM-FPU register)
final static int Count = 5    // current count (uM-FPU register)

Fpu.startWrite();
Fpu.write(Fpu.SELECTA+Total, Fpu.FADD+Count);
Fpu.stop();
```

Selecting the A register is such a common occurrence that the `SELECTA` opcode was defined as `0x00`, so `SELECTA+Total` is the same as just using `Total` by itself. Using this shortcut, line above would be replaced with:

```
Fpu.write(Total, Fpu.FADD+Count);
```

Tutorial Example

Now that we've introduced some of the basic concepts of sending instructions to the uM-FPU, let's go through a tutorial example to get a better understanding of how it all ties together. This example will take a temperature reading from a DS1620 digital thermometer and convert it to Celsius and Fahrenheit.

Most of the data read from devices connected to the Javelin Stamp will return some type of integer value. In this example, the interface routine for the DS1620 reads a 9-bit value and stores it in an integer variable called `rawTemp` on the Javelin Stamp. The value returned by the DS1620 is the temperature in units of 1/2 degrees Celsius. We need to load this value to the uM-FPU and convert it to floating point. The following commands are used:

```
Fpu.write(DegC, Fpu.LOADWORD);
Fpu.writeWord(rawTemp);
Fpu.write(Fpu.FSET);
```

Description:

`DegC`	select DegC as the A register
`LOADWORD`	select register 0 as the B register, load 16-bit value and convert to floating point
`rawTemp`	send 16-bit value
`FSET`	DegC = register 0

The uM-FPU register DegC now contains the value read from the DS1620 (converted to floating point). Since the DS1620 works in units of1/2 degree Celsius, DegC will be divided by 2 to get the degrees in Celsius.

```
Fpu.write(Fpu.LOADBYTE, 2, Fpu.FDIV);
```

Description:

`LOADBYTE`	select register 0 as the B register, load 8-bit value and convert to floating point
`2`	send 8-bit value
`FDIV`	divide DegC by register 0

To get the degrees in Fahrenheit we will use the formula F = C * 1.8 + 32. Since 1.8 and 32 are constant values, they would normally be loaded once in the initialization section of your program and used later in the main program. The value 1.8 is loaded by using the ATOF (ASCII to float) instruction as follows:

```
Fpu.write(F1_8, Fpu.ATOF);
Fpu.writeString("1.8");
Fpu.write(Fpu.FSET);
```

Description:

`F1_8`	select F1_8 as the A register
`ATOF`	select register 0 as the B register, load string and convert to floating point
`"1.8"`	send zero-terminated string
`FSET`	set F1_8 to the value in register 0

The value 32 is loaded using the LOADBYTE instruction as follows:

```
Fpu.write(F32, Fpu.LOADBYTE, 32, Fpu.FSET);
```

Description:

`F32`	select F32 as the A register
`LOADBYTE`	select register 0 as the B register, load 8-bit value and convert to floating point
`32`	send 8-bit value
`FSET`	set F32 to the value in register 0

Now using these constant values we calculate the degrees in Fahrenheit as follows:

```
Fpu.write(DegF, Fpu.FSET+DegC, Fpu.FMUL+F1_8, Fpu.FADD+F32);
```

Description:

`DegF`	select DegF as the A register
`FSET+DegC`	set DegF = DegC
`FMUL+F1_8`	multiply DegF by 1.8
`FADD+F32`	add 32.0 to DegF

Now we print the results. The `Fpu.floatFormat` method is used to convert a floating point value to a formatted string. The first parameter selects the uM-FPU register, and the second parameter specifies the desired format. The tens digit is the total number of characters to display, and the ones digit is the number of digits after the decimal point. The DS1620 has a maximum temperature of 125° Celsius and one decimal point of precision, so we'll use a format of 51. The following example prints the temperature in degrees Fahrenheit.

```
System.out.println(Fpu.floatFormat(DegF, 51));
```

Sample code for this tutorial and a wiring diagram for the DS1620 are shown at the end of this document. The file *demo1.java* is also included with the support software. There is a second file called *demo2.java* that extends this demo to include minimum and maximum temperature calculations. If you have a DS1620 you can wire up the circuit and try out the demos.

Using the uM-FPU Javelin Stamp Packages

Two packages are provided to handle the communication between the Javelin Stamp and the uM-FPU V2 floating point coprocessor, using either a SPI or I^{2}C interface. They are located as follows:

~\lib\com\micromegacorp\math\v2-spi SPI interface
~\lib\com\micromegacorp\math\v2-i2c I^{2}C interface

Each package contains the `Fpu` class which is commented to provide API documentation using Javadoc. One of the following statements should be added to any class that uses the uM-FPU V2 math package.

```
package com.micromegacorp.math.v2-spi;
    or
package com.micromegacorp.math.v2-i2c;
```

With the exception of the interface specific form of the `reset` method, all methods are the same for the SPI and I^{2}C interface, so user programs can be developed using code that is compatible with either interface. The user selects which interface to use by specifying the appropriate package as shown above. All of the device specific code is handled by the `Fpu` class.

Fpu.reset

In order to ensure that the Javelin Stamp 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 registers are reset to the special value NaN (Not a Number) equal to the hexadecimal value `7FC00000`.

Fpu.startWrite

This method is called to start all write transfers.

Fpu.startRead

This method is called to start all read transfers.

Fpu.stop

This method is called to stop a write or read transfer. If a read transfer begins immediately after a write transfer, the `Fpu.stop` is not required. It is also not required if the `Fpu.wait`, `fpu.floatformat`, or `Fpu.longFormat` methods are called, since these methods call `Fpu.stop` internally.

Fpu.wait

This method must be called before issuing any read instruction. It waits until the uM-FPU is ready and the 32-byte instruction buffer is empty.

```
Fpu.wait();
```

```
Fpu.startWrite();
Fpu.write(Fpu.SELECTA, Fpu.XOP, Fpu.READWORD);
int dataWord = Fpu.readWord();
```

Description:
- wait for the uM-FPU to be ready
- send the READWORD instruction
- read a word value and store it in the variable dataWord

The uM-FPU V2 has a 32 byte instruction buffer. In most cases, data will be read back before 32 bytes have been sent to the uM-FPU, but if a calculation requires more than 32 bytes to be sent to the uM-FPU, an `Fpu.wait` call should be made at least every 32 bytes to ensure that the instruction buffer doesn't overflow.

Fpu.write

This method is used to send instructions and data to the uM-FPU. Up to four 8-bit values can be passed as parameters. A `Fpu.startWrite` call must be made at the start of a write transfer, before the first `Fpu.write` call is made.

Fpu.writeWord

This method sends a 16 bit value to the uM-FPU.

Fpu.writeString

This method sends a string to the uM-FPU followed by a zero byte to terminate the string.

Fpu.read

This method is used to read 8 bits of data from the uM-FPU.

Fpu.readWord

This method is used to read a 16 bits of data from the uM-FPU.

Fpu.read32

This method is used to read a 32 bits of data from the uM-FPU. The result is stored in two consecutive elements of an integer array. In most applications this routine is not required, since 32-bit floating point or long integer values are normally left in the uM-FPU registers.

Fpu.readString

This method is used to read a zero terminated string from the uM-FPU. The `Fpu.floatFormat`, `Fpu.longFormat`, and `Fpu.version` methods use this method to return the string. It is rarely called directly by user code.

Fpu.version

This method returns the uM-FPU version string.

Fpu.floatFormat

The floating point value contained in a uM-FPU register is returned as a formatted string. The format parameter is used to specify the desired format. The tens digit specifies the total number of characters to display and the ones digit specifies the number of digits after the decimal point. If the value is too large for the format specified, then asterisks will be displayed. If the number of digits after the decimal points is zero, no decimal point will be displayed. Examples of the display format are as follows:

Value in A register	format	Display format
123.567	61 (6.1)	123.6

```
123.567         62 (6.2)        123.57
123.567         42 (4.2)        *.**
0.9999          20 (2.0)         1
0.9999          31 (3.1)        1.0
```

If the format parameter is omitted, or has a value of zero, the default format is used. Up to eight significant digits will be displayed if required. Very large or very small numbers are displayed in exponential notation. The length of the displayed value is variable and can be from 3 to 12 characters in length. The special cases of NaN (Not a Number), +Infinity, -Infinity, and -0.0 are handled. Examples of the display format are as follows:

```
1.0           NaN           0.0
1.5e20        Infinity      -0.0
3.1415927     -Infinity     1.0
-52.333334    -3.5e-5       0.01
```

Fpu.longFormat

The long integer value contained in a uM-FPU register displayed as a formatted string. The format parameter is used to specify the desired format. A value between 0 and 15 specifies the width of the display field for a signed long integer. The number is displayed right justified. If 100 is added to the format value the value is displayed as an unsigned long integer. If the value is larger than the specified width, asterisks will be displayed. If the width is specified as zero, the length will be variable. Examples of the display format are as follows:

Value in register A	format	Display format
-1	10 (signed 10)	-1
-1	110 (unsigned 10)	4294967295
-1	4 (signed 4)	-1
-1	104 (unsigned 4)	****
0	4 (signed 4)	0
0	0 (unformatted)	0
1000	6 (signed 6)	1000

If the format parameter is omitted, or has a value of zero, the default format is used. The displayed value can range from 1 to 11 characters in length. Examples of the display format are as follows:

```
1
500000
-3598390
```

Loading Data Values to the uM-FPU

There are several instructions for loading integer values to the uM-FPU. These instructions take an integer value as an argument, stores the value in register 0, converts it to floating point, and selects register 0 as the B register. This allows the loaded value to be used immediately by the next instruction.

`LOADBYTE`	Load 8-bit signed integer and convert to floating point
`LOADUBYTE`	Load 8-bit unsigned integer and convert to floating point
`LOADWORD`	Load 16-bit signed integer and convert to floating point
`LOADUWORD`	Load 16-bit unsigned integer and convert to floating point

For example, to calculate Result = Result + 20.0

```
Fpu.write(Result, Fpu.LOADBYTE, 20, Fpu.FADD);
```

Description:

`Result`	select Result as the A register
`LOADBYTE`	select register 0 as the B register, load 8-bit value and convert to floating point
`20`	send 8-bit value
`FADD`	add register 0 to Result

The following instructions take integer value as an argument, stores the value in register 0, converts it to a long integer, and selects register 0 as the B register.

`LONGBYTE`	Load 8-bit signed integer and convert to 32-bit long signed integer
`LONGUBYTE`	Load 8-bit unsigned integer and convert to 32-bit long unsigned integer
`LONGWORD`	Load 16-bit signed integer and convert to 32-bit long signed integer
`LONGUWORD`	Load 16-bit unsigned integer and convert to 32-bit long unsigned integer

For example, to calculate Total = Total / 100

```
Fpu.write(Total, Fpu.XOP, Fpu.LONGBYTE, 100);
Fpu.write(Fpu.LADD);
```

Description:

`Total`	select Total as the A register
`XOP, LONGBYTE`	select register 0 as the B register, load 8-bit value and convert to long integer
`100`	send 8-bit value
`LDIV`	divide Total by register 0

There are several instructions for loading commonly used constants. These instructions load the constant value to register 0, and select register 0 as the B register.

`LOADZERO`	Load the floating point value 0.0 (or long integer 0)
`LOADONE`	Load the floating point value 1.0
`LOADE`	Load the floating point value of e (2.7182818)
`LOADPI`	Load the floating point value of pi (3.1415927)

For example, to set Result = 0.0

```
Fpu.write(Result, Fpu.XOP, Fpu.LOADZERO, Fpu.FSET);
```

Description:

`Result`	select Result as the A register
`XOP, LOADZERO`	select register 0 as the B register, load 0.0
`FSET`	set Result to the value in register 0

There are two instructions for loading 32-bit floating point values to a specified register. This is one of the more efficient ways to load floating point constants, but requires knowledge of the internal representation for floating point numbers (see Appendix B). A handy utility program called *uM-FPU Converter* is available to convert between floating point strings and 32-bit hexadecimal values.

`FWRITEA`	Write 32-bit floating point value to specified register
`FWRITAB`	Write 32-bit floating point value to specified register

For example, to set Angle = 20.0 (the floating point representation for 20.0 is `0x41A00000`)

```
Fpu.write(Fpu.FWRITEA+Angle);
Fpu.writeWord((short)0x41A0);
Fpu.writeWord((short)0x0000);
```

Description:

`FWRITEA+Angle`	select Angle as the A register and load 32-bit value
`0x41,0xA0,0x00,0x00`	send 32-bit value

There are two instructions for loading 32-bit long integer values to a specified register.

`LWRITEA`	Write 32-bit long integer value to specified register
`LWRITAB`	Write 32-bit long integer value to specified register

For example, to set Total = 5000000

```
Fpu.write(Fpu.XOP, Fpu.LWRITEA+Total);
Fpu.writeWord((short)(5000000 >> 16));
Fpu.writeWord((short)(5000000 & 0xFFFF));
```

Description:

`XOP, LWRITEA+Total`	select Total as the A register and load 32-bit value
`(short)(5000000 >> 16)`	send high 16 bits of 32-bit value
`(short)(5000000 & 0xFFFF)`	send low 16 bits of 32-bit value

There are two instructions for converting strings to floating point or long integer values.

`ATOF`	Load ASCII string and convert to floating point
`ATOL`	Load ASCII string and convert to long integer

For example, to set Angle = 1.5885

```
Fpu.write(Angle, Fpu.ATOF);
Fpu.writeString("1.5885");
Fpu.write(Fpu.FSET);
```

Description:

`Angle`	select Angle as the A register
`ATOF`	select register 0 as the B register, load string and convert to floating point
`writeString("1.5885")`	send zero-terminated string
`FSET`	set Angle to the value in register 0

For example, to set Total = 500000

```
Fpu.write(Total, Fpu.ATOL);
Fpu.writeString("5000000");
Fpu.write(Fpu.FSET);
```

Description:

`Total`	select Total as the A register

`ATOL`	select register 0 as the B register, load string and convert to floating point
`"5000000"`	send zero-terminated string
`FSET`	set Total to the value in register 0

The fastest operations occur when the uM-FPU registers are already loaded with values. In time critical portions of code floating point constants should be loaded beforehand to maximize the processing speed in the critical section. With 15 registers available for storage on the uM-FPU, it is often possible to preload all of the required constants. In non-critical sections of code, data and constants can be loaded as required.

Reading Data Values from the uM-FPU

There are two instructions for reading 32-bit floating point values from the uM-FPU.

`READFLOAT`	Reads a 32-bit floating point value from the A register.
`FREAD`	Reads a 32-bit floating point value from the specified register.

The following commands read the floating point value from the A register

```
Fpu.wait();
Fpu.startWrite();
Fpu.write(Fpu.XOP, Fpu.READFLOAT);
Fpu.read32(array);
```

Description:
- wait for the uM-FPU to be ready
- send the READFLOAT instruction
- read the 32-bit value and store it in the first two words of an integer array

There are four instructions for reading integer values from the uM-FPU.

`READBYTE`	Reads the lower 8 bits of the value in the A register.
`READWORD`	Reads the lower 16 bits of the value in the A register.
`READLONG`	Reads a 32-bit long integer value from the A register.
`LREAD`	Reads a 32-bit long integer value from the specified register.

The following commands read the lower 8 bits from the A register

```
Fpu.wait();
Fpu.startWrite();
Fpu.write(Fpu.XOP, Fpu.READBYTE);
dataByte = Fpu.read ();
```

Description:
- wait for the uM-FPU to be ready
- send the READBYTE instruction
- read a byte value and store it in the variable dataByte

Comparing and Testing Floating Point Values

A floating point value can be zero, positive, negative, infinite, or Not a Number (which occurs if an invalid operation is performed on a floating point value). To check the status of a floating point number the `FSTATUS` instruction is sent, and the status byte is returned. The `Fpu` class has a constant defined for each of the status bits as follows:

`ZERO_FLAG`	Zero status bit (0-not zero, 1-zero)
`SIGN_FLAG`	Sign status bit (0-positive, 1-negative)
`NAN_FLAG`	Not a Number status bit (0-valid number, 1-NaN)
`INFINITY_FLAG`	Infinity status bit (0-not infinite, 1-infinite)

For example:

```
Fpu.wait();
Fpu.startWrite();
Fpu.write(Fpu.FSTATUS);
status = Fpu.read();
if ((status & Fpu.ZERO_FLAG) != 0)
    System.out.println("Result is Zero");
else if ((status & Fpu.SIGN_FLAG) != 0)
    System.out.println("Result is Negative");
```

The `FCOMPARE` instruction is used to compare two floating point values. The status bits are set for the results of the operation A – B (The selected A and B registers are not modified). For example, the following commands compare the values in registers Value1 and Value2.

```
Fpu.wait();
Fpu.startWrite();
Fpu.write(Value1, Fpu.SELECTB+Value2, Fpu.FCOMPARE);
status = Fpu.read();
if ((status & Fpu.ZERO_FLAG) != 0)
    System.out.println("Value1 = Value2");
else if ((status & Fpu.SIGN_FLAG) != 0)
    System.out.println("Value1 < Value2");
else
    System.out.println("Value1 > Value2");
```

Comparing and Testing Long Integer Values

A long integer value can be zero, positive, or negative. To check the status of a long integer number the `LSTATUS` instruction is sent, and the returned byte is stored in the `status` variable. A bit definition is provided for each status bit in the `status` variable. They are as follows:

`ZERO_FLAG`	Zero status bit (0-not zero, 1-zero)
`SIGN_FLAG`	Sign status bit (0-positive, 1-negative)

For example:

```
Fpu.wait();
Fpu.startWrite();
Fpu.write(Fpu.XOP, Fpu.LSTATUS);
status = Fpu.read();
if ((status & Fpu.ZERO_FLAG) != 0)
    System.out.println("Result is Zero");
else if ((status & Fpu.SIGN_FLAG) != 0)
    System.out.println("Result is Negative");
```

The `LCOMPARE` and `LUCOMPARE` instructions are used to compare two long integer values. The status bits are set for the results of the operation A – B (The selected A and B registers are not modified). `LCOMPARE` does a signed compare and `LUCOMPARE` does an unsigned compare. For example, the following commands compare the values in registers Value1 and Value2.

```
Fpu.wait();
Fpu.startWrite();
Fpu.write(Value1, Fpu.SELECTB+Value2, Fpu.XOP, Fpu.LCOMPARE);
status = Fpu.read();
if ((status & Fpu.ZERO_FLAG) != 0)
    System.out.println("Value1 = Value2");
else if ((status & Fpu.SIGN_FLAG) != 0)
    System.out.println("Value1 < Value2");
else
    System.out.println("Value1 > Value2");
```

Left and Right Parenthesis

Mathematical equations are often expressed with parenthesis to define the order of operations. For example Y = (X-1) / (X+1). The `LEFT` and `RIGHT` parenthesis instructions provide a convenient means of allocating temporary values and changing the order of operations.

When a `LEFT` parenthesis instruction is sent, the current selection for the A register is saved and the A register is set to reference a temporary register. Operations can now be performed as normal with the temporary register selected as the A register. When a `RIGHT` parenthesis instruction is sent, the current value of the A register is copied to register 0, register 0 is selected as the B register, and the previous A register selection is restored. The value in register 0 can be used immediately in subsequent operations. Parenthesis can be nested for up to five levels. In most situations, the user's code does not need to select the A register inside parentheses since it is selected automatically by the `LEFT` and `RIGHT` parentheses instructions.

In the following example the equation Z = sqrt(X**2 + Y**2) is calculated. Note that the original values of X and Y are retained.

```
final static int Xvalue = 1   // X value (uM-FPU register 1)
final static int Yvalue = 2   // Y value (uM-FPU register 2)
final static int Zvalue = 3   // Z value (uM-FPU register 3)

Fpu.startWrite();
Fpu.write(Zvalue, Fpu.FSET+Xvalue, Fpu.FMUL+Xvalue);
Fpu.write(Fpu.XOP, Fpu.LEFT, Fpu.FSET+Yvalue, Fpu.FMUL+Yvalue);
Fpu.write(Fpu.XOP, Fpu.RIGHT, Fpu.FADD, Fpu.SQRT);
```

Description:

`Zvalue`	select Zvalue as the A register
`FSET+Xvalue`	Zvalue = Xvalue
`FMUL+Xvalue`	Zvalue = Zvalue * Xvalue (i.e. X**2)
`XOP, LEFT`	save current A register selection, select temporary register as A register (temp)
`FSET+Yvalue`	temp = Yvalue
`FMUL+Yvalue`	temp = temp * Yvalue (i.e. Y**2)
`XOP, RIGHT`	store temp to register 0, select Zvalue as A register (previously saved selection)
`FADD`	add register 0 to Zvalue (i.e. X**2 + Y**2)
`SQRT`	take the square root of Zvalue

The following example shows Y = 10 / (X + 1):

```
Fpu.startWrite();
Fpu.write(Yvalue, Fpu.LOADBYTE, 10, Fpu.FSET);
Fpu.write(Fpu.XOP, Fpu.LEFT, Fpu.FSET+Xvalue);
Fpu.write(Fpu.XOP, Fpu.LOADONE, Fpu.FADD);
Fpu.write(Fpu.XOP, Fpu.RIGHT, Fpu.FDIV);
```

Description:

`Yvalue`	select Yvalue as the A register
`LOADBYTE, 10`	load the value 10 to register 0, convert to floating point, select register 0 as the B register
`FSET`	Yvalue = 10.0
`XOP, LEFT`	save current A register selection, select temporary register as A register (temp)
`FSET+Xvalue`	temp = Xvalue
`XOP, LOADONE`	load 1.0 to register 0 and select register 0 as the B register
`FADD`	temp = temp + 1 (i.e. X+1)
`XOP, RIGHT`	store temp to register 0, select Yvalue as A register (previously saved selection)
`FDIV`	divide Yvalue by the value in register 0

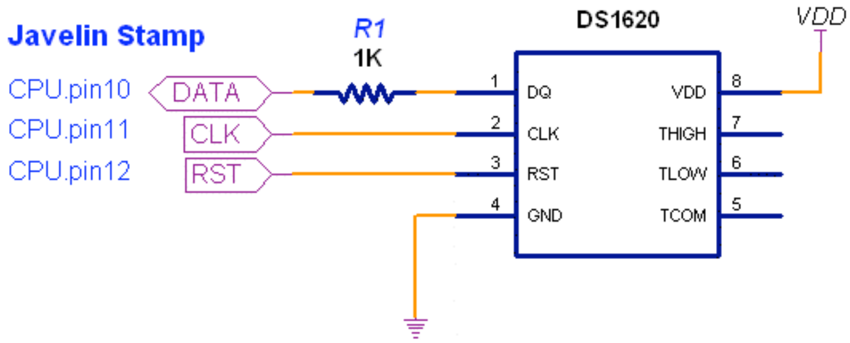
Further Information

The following documents are also available:

uM-FPU V2 Datasheet	provides hardware details and specifications
uM-FPU V2 Instruction Reference	provides detailed descriptions of each instruction
uM-FPU Application Notes	various application notes and examples

Check the Micromega website at www.micromegacorp.com

DS1620 Connections for Demo 1



Sample Code for Tutorial (Demo1.java)

```
import com.micromegacorp.math.v2_spi.*;   // (use one of the uM-FPU packages)
//import com.micromegacorp.math.v2_i2c.*;
import stamp.core.*;
import stamp.peripheral.sensor.temperature.DS1620;

// This program demonstrates how to use the uM-FPU V2 floating point coprocessor
// connected to the Javelin Stamp using either a 2-wire SPI or I2C interface.
// It takes temperature readings from a DS1620 digital thermometer, converts
// them to floating point and displays them in degrees Celsius and degrees
// Fahrenheit.
public class Demo1 {

  final static int DS_DATA = CPU.pin10; // DS1620 data pin
  final static int DS_CLK  = CPU.pin11; // DS1620 clock pin
  final static int DS_RST  = CPU.pin12; // DS1620 reset/enable pin

  //-------------------- uM-FPU register definitions -------------------------

  final static int DegC = 1;            // degrees Celsius
  final static int DegF = 2;            // degrees Fahrenheit
  final static int F1_8 = 3;            // constant 1.8
  final static int F32  = 4;            // constant 32.0

  //-------------------- main routine -----------------------------------------

  public static void main() {
    int rawTemp;

    // display program name
    System.out.println("\u0010Demo1");

    // reset the uM-FPU and print version string
    if (!Fpu.reset()) {
      System.out.println("uM-FPU not responding.");
      return;
    }
    else
      System.out.println(Fpu.version());

    // get a DS1620 object and initialize
    DS1620 ds = new DS1620(DS_DATA, DS_CLK, DS_RST);
```

```
    CPU.delay(10000);

    // store constant values (1.8 and 32.0)
    Fpu.startWrite();
    Fpu.write(F1_8, Fpu.ATOF);
    Fpu.writeString("1.8");
    Fpu.write(Fpu.FSET);
    Fpu.write(F32, Fpu.LOADBYTE, 32, Fpu.FSET);
    Fpu.stop();

    // loop forever, read and display temperature
    while (true) {
      // get temperature reading from DS1620
      rawTemp = ds.getTempRaw();

      // send to uM-FPU and convert to floating point
      Fpu.startWrite();
      Fpu.write(DegC, Fpu.LOADWORD);
      Fpu.writeWord(rawTemp);
      Fpu.write(Fpu.FSET);

      // divide by 2 to get degrees Celsius
      Fpu.write(Fpu.LOADBYTE, 2, Fpu.FDIV);

      // degF = degC * 1.8 + 32
      Fpu.write(DegF, Fpu.FSET+DegC, Fpu.FMUL+F1_8, Fpu.FADD+F32);
      Fpu.stop();

      // display degrees Celsius
      System.out.print("\n\rDegrees C: ");
      System.out.println(Fpu.floatFormat(DegC, 51));

       // display degrees Fahrenheit
      System.out.print("Degrees F: ");
      System.out.println(Fpu.floatFormat(DegF, 51));

      // delay about 2 seconds, then get the next reading
      CPU.delay(21000);
    }
  }
} // end class
```

Appendix A
uM-FPU V2 Instruction Summary (Javelin Stamp definitions)

Opcode Name	Data Type	Opcode	Arguments	Returns	B Reg	Description
SELECTA		0x				Select A register
SELECTB		1x			x	Select B register
FWRITEA	Float	2x	yyyy zzzz			Select A register, Write floating point value to A register
FWRITEB	Float	3x	yyyy zzzz		x	Select B register, Write floating point value to B register
FREAD	Float	4x		yyyy zzzz		Read register
FSET/LSET	Either	5x				Select B register, A = B
FADD	Float	6x			x	Select B register, A = A + B
FSUB	Float	7x			x	Select B register, A = A - B
FMUL	Float	8x			x	Select B register, A = A * B
FDIV	Float	9x			x	Select B register, A = A / B
LADD	Long	Ax			x	Select B register, A = A + B
LSUB	Long	Bx			x	Select B register, A = A -B
LMUL	Long	Cx			x	Select B register, A = A * B
LDIV	Long	Dx			x	Select B register, A = A / B Remainder stored in register 0
SQRT	Float	E0				A = sqrt(A)
LOG	Float	E1				A = ln(A)
LOG10	Float	E2				A = log(A)
EXP	Float	E3				A = e ** A
EXP10	Float	E4				A = 10 ** A
SIN	Float	E5				A = sin(A) radians
COS	Float	E6				A = cos(A) radians
TAN	Float	E7				A = tan(A) radians
FLOOR	Float	E8				A = nearest integer <= A
CEIL	Float	E9				A = nearest integer >= A
ROUND	Float	EA				A = nearest integer to A
NEGATE	Float	EB				A = -A
ABS	Float	EC				A = |A|
INVERSE	Float	ED				A = 1 / A
DEGREES	Float	EE				Convert radians to degrees A = A / (PI / 180)
RADIANS	Float	EF				Convert degrees to radians A = A * (PI / 180)
SYNC		F0		5C		Synchronization
FLOAT	Long	F1			0	Copy A to register 0 Convert long to float
FIX	Float	F2			0	Copy A to register 0 Convert float to long
FCOMPARE	Float	F3		ss		Compare A and B (floating point)
LOADBYTE	Float	F4	bb		0	Write signed byte to register 0 Convert to float
LOADUBYTE	Float	F5	bb		0	Write unsigned byte to register 0 Convert to float
LOADWORD	Float	F6	wwww		0	Write signed word to register 0 Convert to float
LOADUWORD	Float	F7	wwww		0	Write unsigned word to register 0 Convert to float

READSTR		F8		aa … 00		Read zero terminated string from string buffer
ATOF	Float	F9	aa … 00		0	Convert ASCII to float Store in register 0
FTOA	Float	FA	ff			Convert float to ASCII Store in string buffer
ATOL	Long	FB	aa … 00		0	Convert ASCII to long Store in register 0
LTOA	Long	FC	ff			Convert long to ASCII Store in string buffer
FSTATUS	Float	FD		ss		Get floating point status of A
XOP		FE				Extended opcode prefix (extended opcodes are listed below)
NOP		FF				No Operation
FUNCTION		FE0n FE1n FE2n FE3n			0	User defined functions 0-15 User defined functions 16-31 User defined functions 32-47 User defined functions 48-63
IF_FSTATUSA	Float	FE80	ss			Execute user function code if FSTATUSA conditions match
IF_FSTATUSB	Float	FE81	ss			Execute user function code if FSTATUSB conditions match
IF_FCOMPARE	Float	FE82	ss			Execute user function code if FCOMPARE conditions match
IF_LSTATUSA	Long	FE83	ss			Execute user function code if LSTATUSA conditions match
IF_LSTATUSB	Long	FE84	ss			Execute user function code if LSTATUSB conditions match
IF_LCOMPARE	Long	FE85	ss			Execute user function code if LCOMPARE conditions match
IF_LUCOMPARE	Long	FE86	ss			Execute user function code if LUCOMPARE conditions match
IF_LTST	Long	FE87	ss			Execute user function code if LTST conditions match
TABLE	Either	FE88				Table Lookup (user function)
POLY	Float	FE89				Calculate n^{th} degree polynomial (user function)
READBYTE	Long	FE90		bb		Get lower 8 bits of register A
READWORD	Long	FE91		wwww		Get lower 16 bits of register A
READLONG	Long	FE92		yyyy zzzz		Get long integer value of register A
READFLOAT	Float	FE93		yyyy zzzz		Get floating point value of register A
LINCA	Long	FE94				A = A + 1
LINCB	Long	FE95				B = B + 1
LDECA	Long	FE96				A = A - 1
LDECB	Long	FE97				B = B - 1
LAND	Long	FE98				A = A AND B
LOR	Long	FE99				A = A OR B
LXOR	Long	FE9A				A = A XOR B
LNOT	Long	FE9B				A = NOT A
LTST	Long	FE9C	ss			Get the status of A AND B
LSHIFT	Long	FE9D				A = A shifted by B bit positions
LWRITEA	Long	FEAx	yyyy zzzz			Write register and select A
LWRITEB	Long	FEBx	yyyy zzzz		x	Write register and select B
LREAD	Long	FECx		yyyy zzzz		Read register
LUDIV	Long	FEDx			x	Select B register, A = A / B (unsigned) Remainder stored in register 0
POWER	Float	FEE0				A = A raised to the power of B
ROOT	Float	FEE1				A = the Bth root of A

MIN	Float	FEE2				A = minimum of A and B
MAX	Float	FEE3				A = maximum of A and B
FRACTION	Float	FEE4			0	Load Register 0 with the fractional part of A
ASIN	Float	FEE5				A = asin(A) radians
ACOS	Float	FEE6				A = acos(A) radians
ATAN	Float	FEE7				A = atan(A) radians
ATAN2	Float	FEE8				A = atan(A/B)
LCOMPARE	Long	FEE9		ss		Compare A and B (signed long integer)
LUCOMPARE	Long	FEEA		ss		Compare A and B (unsigned long integer)
LSTATUS	Long	FEEB		ss		Get long status of A
LNEGATE	Long	FEEC				A = -A
LABS	Long	FEED				A = |A|
LEFT		FEEE				Left parenthesis
RIGHT		FEEF			0	Right parenthesis
LOADZERO	Float	FEF0			0	Load Register 0 with Zero
LOADONE	Float	FEF1			0	Load Register 0 with 1.0
LOADE	Float	FEF2			0	Load Register 0 with e
LOADPI	Float	FEF3			0	Load Register 0with pi
LONGBYTE	Long	FEF4	bb		0	Write signed byte to register 0 Convert to long
LONGUBYTE	Long	FEF5	bb		0	Write unsigned byte to register 0 Convert to long
LONGWORD	Long	FEF6	wwww		0	Write signed word to register 0 Convert to long
LONGUWORD	Long	FEF7	wwww		0	Write unsigned word to register 0 Convert to long
IEEEMODE		FEF8				Set IEEE mode (default)
PICMODE		FEF9				Set PIC mode
CHECKSUM		FEFA			0	Calculate checksum for uM-FPU code
BREAK		FEFB				Debug breakpoint
TRACEOFF		FEFC				Turn debug trace off
TRACEON		FEFD				Turn debug trace on
TRACESTR		FEFE	aa … 00			Send debug string to trace buffer
VERSION		FEFF				Copy version string to string buffer

Notes:

Data Type	data type required by opcode
Opcode	hexadecimal opcode value
Arguments	additional data required by opcode
Returns	data returned by opcode
B Reg	value of B register after opcode executes
x	register number (0-15)
n	function number (0-63)
yyyy	most significant 16 bits of 32-bit value
zzzz	least significant 16 bits of 32-bit value
ss	status byte
bb	8-bit value
wwww	16-bit value
aa … 00	zero terminated ASCII string

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:

S	Exponent	Mantissa
31	30 … 23	22 … 0

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
```