Micromega Corporation

Using uM-FPU V2 with the Comfile PICBASIC Microcontrollers

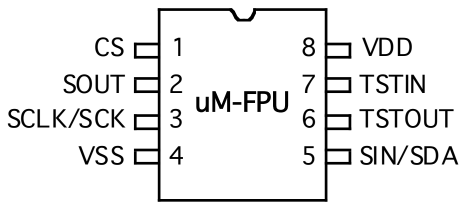
Introduction

The uM-FPU is a 32-bit floating point coprocessor that is easily interfaced with the PICBASIC family of microcontrollers from Comfile Technology. The uM-FPU supports 32-bit floating point and 32-bit long integer operations and can be connected using either an I^2C or SPI interface.

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. 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
  - 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 SCK	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

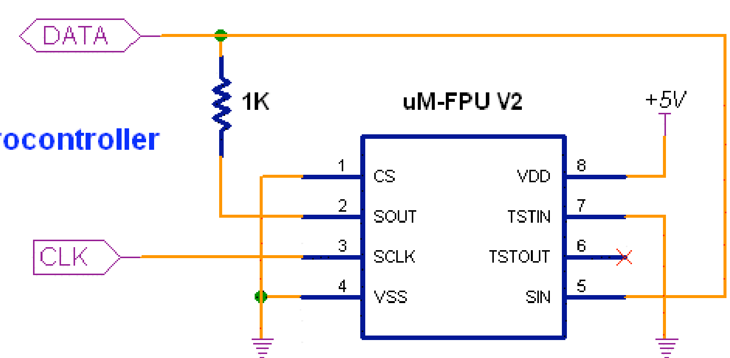
Connecting the uM-FPU to the Comfile PICBASIC using SPI

The 2-wire SPI connection uses a clock signal and a bidirectional data signal. Examples of the pin settings are as follows:

Pin	2-wire SPI
`FPU_CLK`	`I/O15`
`FPU_DATIN`	`I/O14`
`FPU_DATOUT`	`I/O14`

The settings of these pins can be changed to suit your application. By default, the uM-FPU chip is always selected, so the `FPU_CLK` and `FPU_DATIN/FPU_DATIN` pins should not be used for other connections as this will likely result in loss of synchronization between the PICBASIC microprocessor and the uM-FPU coprocessor.

2-wire SPI Connection



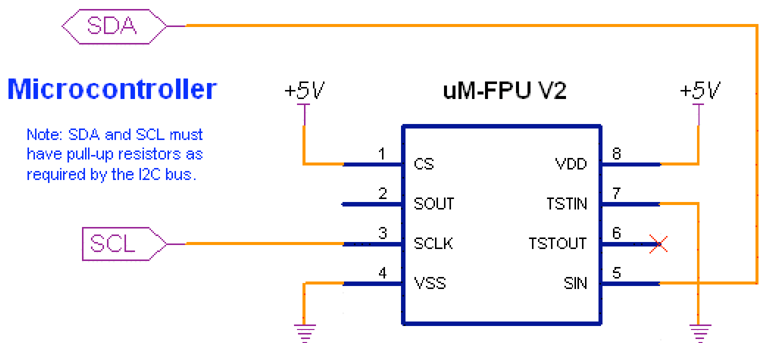
If a 2-wire SPI interface is used, the SOUT and SIN pins should not be connected directly together, ***they must be connected through a 1K resistor.*** The microcontroller data pin is connected to the SIN pin. See the uM-FPU datasheet for further description of the SPI interface.

Connecting the uM-FPU to the Comfile PICBASIC using I^2C

The uM-FPU V2 can also be connected using an I^2C interface. The default slave address for the uM-FPU is 0xC8 (LSB is the R/W bit, e.g. 0xC8 for write, 0xC9 for read). See the uM-FPU datasheet for further description of the I^2C interface.

FPU_SCL	I^2C clock
FPU_SDA	I^2C data

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 Comfile PICBASIC commands 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 registers. 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, with 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*.

Instructions and data are sent to the uM-FPU by loading the `dataByte` variable with the byte value to send and calling the `fpu_sendOp` routine. For example:

```
dataByte = FADD+5
gosub fpu_sendOp
```

All instructions start with an opcode that tells the uM-FPU which operation to perform. 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:

```
dataByte = SQRT
gosub fpu_sendOp
```

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 always `&hFE`, defined as `XOP`. To use an extended opcode, you use the `fpu_sendXop` routine. This routine sends an `XOP` byte first, followed by opcode specified by the `dataByte` variable. For example:

```
dataByte = ATAN
gosub fpu_sendXop
```

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.

```
dataByte = FADD
gosub fpu_sendOp
```

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.

```
dataByte = FADD+5
gosub fpu_sendOp
```

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:

```
dataByte = SELECTA+2          ' select register 2 as A register
gosub fpu_sendOp
dataByte = FADD+5             ' select register 5 as B register and
gosub fpu_sendOp              '  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:

```
const Total = 2               'total amount  (uM-FPU register)
const Count = 5               'current count (uM-FPU register)

dataByte = SELECTA+Total      ' select Total as A register
gosub fpu_sendOp
dataByte = FADD+Count         ' select Count as B register and
gosub fpu_sendOp              '  calculate A = A + B
```

Selecting the A register is such a common occurrence, it was defined as opcode &h0x. The definition for `SELECTA` is `&h00`, so `SELECTA+Total` is the same as just using `Total` by itself. Using this shortcut, the same example would now be:

```
dataByte = Total              ' select Total as A register
gosub fpu_sendOp
dataByte = FADD+Count         ' select Count as B register and
gosub fpu_sendOp              '  calculate A = A + B
```

Tutorial Examples

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 PICBASIC 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 a Word variable called rawTemp. 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:

```
dataByte = DegC                  ' select DegC as A register
gosub fpu_sendOp
dataByte = LOADWORD              ' load rawTemp to register 0,
gosub fpu_sendOp                 '  convert to floating point,
dataByte = rawTemp.H             '  select register 0 as B register
gosub fpu_sendByte
dataByte = rawTemp.L
gosub fpu_sendByte
dataByte = FSET                  ' DegC = register 0 (i.e. set to the
gosub fpu_sendOp                 '  floatiing point value of rawTemp)
```

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.

```
dataByte = LOADBYTE              ' load the value 2 to register 0,
gosub fpu_sendOp                 '  convert to floating point,
dataByte = 2                     '  select register 0 as the B register
gosub fpu_sendByte
dataByte = FDIV                  ' divide DegC by register 0
gosub fpu_sendOp                 '  (i.e. divide by 2)
```

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:

```
dataByte = F1_8                  ' select F1_8 as A register
gosub fpu_sendOp
dataByte = ATOF                  ' load ASCII string to uM-FPU,
gosub fpu_sendOp                 '  convert to floating point,
dataByte = "1"                   '  store in register 0
gosub fpu_sendByte               '  select register 0 as B register
dataByte = "."
gosub fpu_sendByte
dataByte = "8"
gosub fpu_sendByte
dataByte = 0
gosub fpu_sendByte
dataByte = FSET                  ' F1_8 = register 0 (1.8)
gosub fpu_sendOp
```

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

```
dataByte = F32                   ' select F32 as A register
gosub fpu_sendOp
dataByte = LOADBYTE              ' load the value 32 to register 0,
gosub fpu_sendOp                 '  convert to floating point,
dataByte = 32                    '  select register 0 as the B register
gosub fpu_sendByte
dataByte = FSET                  ' F32 = register 0 (32.0)
gosub fpu_sendOp
```

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

```
dataByte = degF                   ' select DegF as A register
gosub  fpu_sendOp
dataByte = FSET+DegC              ' set DegF = DegC
gosub fpu_sendOp
dataByte = FMUL+F1_8              ' multiply DegF by 1.8
gosub fpu_sendOp
dataByte = FADD+F32               ' add 32.0 to DegF
gosub fpu_sendOp
```

Now we print the results. There are support routines provided for printing floating point numbers. Print_Float prints an unformatted floating point value and displays up to eight digits of precision. Print_FloatFormat prints a formatted floating point number. We'll use Print_FloatFormat to display the results. The `format` variable is used to select 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. Before calling the print routine the uM-FPU register is selected and the `format` variable is set. The following example prints the temperature in degrees Fahrenheit.

```
dataByte = degF                   ' select DegF as A register
gosub  fpu_sendOp
format = 51                       ' print DegF in 5.1 format
gosub print_floatFormat
```

Sample code for this tutorial and a wiring diagram for the DS1620 are shown at the end of this document. The file *demo1.bas* is also included with the support software. There is a second file called *demo2.bas* 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.

uM-FPU Support Software for the Comfile PICBASIC Compiler

A template file containing all of the definitions and support code required for communicating with the uM-FPU is provided for the SPI interface.

`umfpu-spi.bas` provides support for a SPI connection

These files can be used directly as the starting point for a new program, or the definitions and support code can be copied from this file to another program. They contain the following:

- pin definitions for the uM-FPU
- opcode definitions for all uM-FPU instructions
- various definitions for the Word variable used by the support routines
- a sample program with a place to insert your application code
- the support routines described below

fpu_reset

To ensure that the PICBASIC microcontroller and the uM-FPU coprocessor are synchronized, a reset call must be done at the start of every program. The `fpu_reset` routine resets the uM-FPU, confirms communications, and returns the Sync character (`&h5C`) if successful.

fpu_sendOp, fpu_sendByte

Sends the opcode or data value contained in the `dataByte` variable to the uM-FPU.

fpu_sendXop

Sends an `XOP` byte followed by the opcode contained in the `dataByte` variable to the uM-FPU. Used to send two-byte opcodes.

fpu_readByte

Reads a byte of data from the uM-FPU and returns the value in the `dataByte` variable.

fpu_wait

The uM-FPU must have completed all calculations and be ready to return the data before sending an instruction that reads data from the uM-FPU. The `fpu_wait` routine checks the status of the uM-FPU and waits until it is ready. The print routines check the ready status, so it isn't necessary to call `fpu_wait` before calling a print routine. If your program reads directly from the uM-FPU, a call to `fpu_wait` must be made prior to sending the read instruction. An example of reading a byte value is as follows:

```
gosub fpu_wait                                  ' wait until uM-FPU is ready
dataByte = READBYTE                             ' read value
gosub fpu_sendXop
gosub fpu_readByte
```

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. If a long calculation is done that 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.

print_version

Prints the uM-FPU version string to the PC screen using the DEBUG command.

print_float

The value in register A is displayed on the serial terminal as a floating point value using the `serout` command. 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
```

print_floatFormat

The value in register A is displayed on the serial terminal as a formatted floating point value using the `serout` command. The `format` variable 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

print_long

The value in register A is displayed on the serial terminal as a signed long integer using the `serout` command. The displayed value can range from 1 to 11 characters in length. Examples of the display format are as follows:

```
1
500000
-3598390
```

print_longFormat

The value in register A is displayed on the serial terminal as a formatted long integer using the `serout` command. The `format` variable 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

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

```
dataByte = Result                ' select Result as A register
gosub fpu_sendOp
dataByte = LOADBYTE              ' load the value 20 to register 0,
gosub fpu_sendOp                 '  convert to floating point,
dataByte = 20                    '  select register 0 as the B register
gosub fpu_sendByte
dataByte = FADD                  ' add register 0 to Result
gosub fpu_sendOp                 ' (Result = Result + 20.0)
```

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

```
dataByte = Total                 ' select Total as A register
gosub fpu_sendOp
dataByte = LONGBYTE              ' load the value 100 to register 0,
gosub fpu_sendXop                '  convert to floating point,
dataByte = 100                   '  select register 0 as the B register
gosub fpu_sendByte
dataByte = LDIV                  ' divide Total by register 0
gosub fpu_sendOp                 ' (Total = Total / 100.0)
```

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

```
dataByte = Result                  ' select Result as A register
gosub fpu_sendOp
dataByte = LOADZERO                ' load the value 0.0 to register 0,
gosub fpu_sendXop
dataByte = FSET                    ' set Result to value in register 0
gosub fpu_sendOp                   ' (Result = 0.0)
```

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.

`WRITEA`	Write 32-bit floating point value to specified register
`WRITAB`	Write 32-bit floating point value to specified register

For example, to set Angle = 20.0 (the floating point representation for 20.0 is &h41A00000)

```
dataByte = WRITEA+Angle            ' select Angle as A register, and
gosub fpu_sendOp                   '  load 32-bit value
dataByte = &h41
gosub fpu_sendByte
dataByte = &hA0
gosub fpu_sendByte
dataByte = &h00
gosub fpu_sendByte
dataByte = &h00
gosub fpu_sendByte
```

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 = 500000

```
dataByte = LWRITEA+Total           ' select Total as A register, and
gosub fpu_sendXop                  '  load 32-bit value
dataByte = &h00
gosub fpu_sendByte
dataByte = &h07
gosub fpu_sendByte
dataByte = &hA1
gosub fpu_sendByte
dataByte = &h20
gosub fpu_sendByte
```

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

```
dataByte = Angle                   ' select Angle as A register
gosub fpu_sendOp
dataByte = ATOF                    ' load ASCII string to uM-FPU,
gosub fpu_sendOp                   '  convert to floating point,
dataByte = "1"                     '  store in register 0
gosub fpu_sendByte                 '  select register 0 as B register
dataByte = "."
gosub fpu_sendByte
dataByte = "5"
```

```
        gosub fpu_sendByte
        dataByte = "8"
        gosub fpu_sendByte
        dataByte = "8"
        gosub fpu_sendByte
        dataByte = "5"
        gosub fpu_sendByte
        dataByte = 0                      ' (zero terminator)
        gosub fpu_sendByte
        dataByte = FSET                   ' Angle = register 0 (1.5885)
        gosub fpu_sendOp
```

For example, to set Total = 500000

```
        dataByte = Total                  ' select Total as A register
        gosub fpu_sendOp
        dataByte = ATOF                   ' load ASCII string to uM-FPU,
        gosub fpu_sendOp                  '  convert to long integer,
        dataByte = "5"                    '  store in register 0
        gosub fpu_sendByte                '  select register 0 as B register
        dataByte = "0"
        gosub fpu_sendByte
        dataByte = "0"
        gosub fpu_sendByte
        dataByte = "0"
        gosub fpu_sendByte
        dataByte = "0"
        gosub fpu_sendByte
        dataByte = "0"
        gosub fpu_sendByte
        dataByte = “0”
        gosub fpu_sendByte
        dataByte = 0                      ' (zero terminator)
        gosub fpu_sendByte
        dataByte = LSET                   ' Angle = register 0 (1.5885)
        gosub fpu_sendOp
```

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 instruction 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

```
        gosub fpu_wait                    ' wait until uM-FPU is ready
        dataByte = READFLOAT              ' read floating point value
        gosub fpu_sendXop
        gosub fpu_readByte                ' read 32-bit value
        byte1 = dataByte
        gosub fpu_readByte
        byte2 = dataByte
        gosub fpu_readByte
        byte3 = dataByte
        gosub fpu_readByte
        byte4 = dataByte
```

There are four instruction 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

```
gosub fpu_wait                    ' wait until uM-FPU is ready
dataByte = READBYTE               ' read floating point value
gosub fpu_sendXop
gosub fpu_readByte                ' read 8-bit value (to 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 returned byte is read. The following definitions are provided for checking the floating point status bits:

`IS_ZERO`	Plus zero
`IS_NZERO`	Minus zero
`IS_NEGATIVE`	Negative
`IS_NAN`	Not-a-Number
`IS_PINF`	Plus infinity
`IS_NINF`	Minus infinity

The `FSTATUS` command is used to check the status of a floating point number. For example:

```
gosub fpu_wait                       ' wait until uM-FPU is ready
dataByte = FSTATUS                   ' get floating point status of A register
gosub fpu_sendOp
gosub fpu_readByte                   ' read the status byte

if dataByte = IS_ZERO or dataByte = IS_NZERO then goto zeroValue
if dataByte = IS_NEGATIVE then goto negativeValue
  serout Terminal, Sbaud, Smode, Sint, [value is positive"]
  …
negativeValue:
 serout Terminal, Sbaud, Smode, Sint, [value is negative"]
  …
zeroValue:
  serout Terminal, Sbaud, Smode, Sint, [value is zero"]
```

The `FCOMPARE` command 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:

```
gosub fpu_wait                       ' wait until uM-FPU is ready
dataByte = FCOMPARE                  ' get floating point compare of A and B
gosub fpu_sendOp
gosub fpu_readByte                   ' read the status byte

if dataByte = IS_ZERO then goto sameAs
if dataByte = IS_NEGATIVE then goto lessThan
  serout Terminal, Sbaud, Smode, Sint, [“A > B"]
  …
lessThan:
 serout Terminal, Sbaud, Smode, Sint, ["A < B"]
  …
sameAs:
 serout Terminal, Sbaud, Smode, Sint, ["A = B"]
```

…

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. The following symbols define the long status bits:

`IS_ZERO`	Plus zero
`IS_NEGATIVE`	Negative

The `LSTATUS` command is used to check the status of a long integer number. For example:

```
gosub fpu_wait                      ' wait until uM-FPU is ready
dataByte = LSTATUS                  ' get long integer status of A register
gosub fpu_sendOp
gosub fpu_readByte                  ' read the status byte

if dataByte = IS_ZERO then goto zeroValue
if dataByte = IS_NEGATIVE then goto negativeValue

  serout Terminal, Sbaud, Smode, Sint, ["value is positive"]
  …
negativeValue:
  serout Terminal, Sbaud, Smode, Sint, ["value is negative"]
  …
zeroValue:
 serout Terminal, Sbaud, Smode, Sint, ["value is zero"]
```

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

```
gosub fpu_wait                      ' wait until uM-FPU is ready
dataByte = LCOMPARE                 ' get long integer compare of A and B
gosub fpu_sendOp
gosub fpu_readByte                  ' read the status byte

if dataByte = IS_ZERO then goto sameAs
if dataByte = IS_NEGATIVE then goto lessThan
 serout Terminal, Sbaud, Smode, Sint, ["A > B"]
  …
lessThan:
 serout Terminal, Sbaud, Smode, Sint, ["A < B"]
  …
sameAs:
 serout Terminal, Sbaud, Smode, Sint, ["A = B"]
  …
```

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.

```
const Xvalue = 1                        ' X value (uM-FPU register 1)
const Yvalue = 2                        ' Y value (uM-FPU register 2)
const Zvalue = 3                        ' Z value (uM-FPU register 3)

   dataByte = Zvalue                    ' select Zvalue as the A register
   gosub fpu_sendByte

   dataByte = FSET+Xvalue               ' Zvalue = Xvalue
   gosub fpu_sendOp

   dataByte = FMUL+Xvalue               ' Zvalue = Zvalue * Xvalue (i.e. X**2)
   gosub fpu_sendOp

   dataByte = LEFT                      ' save current A register selection and select
   gosub fpu_sendXop                    '  temporary register as A register (temp)

   dataByte = FSET+Yvalue               ' temp = Yvalue
   gosub fpu_sendOp

   dataByte = FMUL+Yvalue               ' temp = temp * Yvalue (i.e. Y**2)
   gosub fpu_sendOp

   dataByte = RIGHT                     ' store temp to register 0, and select Zvalue
   gosub fpu_sendXop                    '  as A register (previously saved selection)

   dataByte = FADD                      ' add register 0 to Zvalue (i.e. X**2 + Y**2)
   gosub fpu_sendOp

   dataByte = SQRT                      ' take the square root of Zvalue
   gosub fpu_sendOp
```

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

```
   dataByte = Yvalue                    ' select Yvalue as the A register
   gosub fpu_sendOp

   dataByte = LOADBYTE                  ' load the value 10 to register 0,
   gosub fpu_sendOp
   dataByte = 10                        '  select register 0 as the B register
   gosub fpu_sendByte

   dataByte = FSET                      ' Yvalue = 10.0
   gosub fpu_sendOp
```

```
    dataByte = LEFT                     ' save current A register selection and select
    gosub fpu_sendXop                   '  temporary register as A register (temp)

    dataByte = FSET+Xvalue              ' temp = Xvalue
    gosub fpu_sendOp

    dataByte = LOADONE                  ' load 1.0 to register 0,
    gosub fpu_sendXop                   '  select register 0 as the B register

    dataByte = FADD                     '  temp = temp + 1 (i.e. X+1)
    gosub fpu_sendByte

    dataByte = RIGHT                    ' store temp to register 0, and select Yvalue
    gosub fpu_sendXop                   '  as A register (previously saved selection)

    dataByte = FDIV                     ' divide Yvalue by the value in register 0
    gosub fpu_sendOp
```

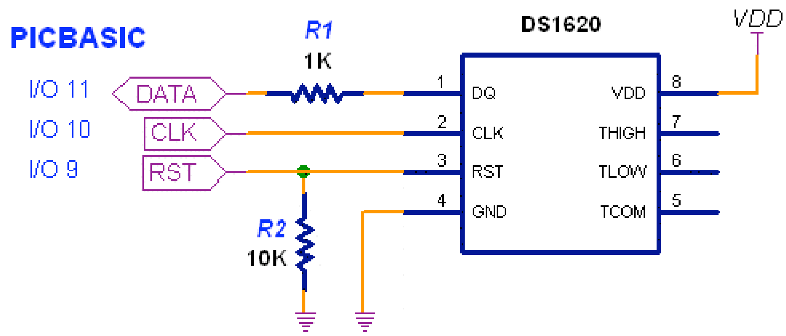
Further Information

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

Check the Micromega website at www.micromegacorp.com for up-to-date information.

DS1620 Connections for Demo 1



Sample Code for Tutorial (Demo1.bas)

```
' This program demonstrates the use of the uM-FPU V2 floating point coprocessor
' with the Comfile PICBASIC microcontroller using an SPI interface. It takes
' temperature readings from a DS1620 digital thermometer, converts them to
' floating point and displays them in degrees Celsius and degrees Fahrenheit.

'-------------------- DS1620 pin definitions ----------------------------------

const  DS_RST         = 9                        ' DS1620 reset/enable
const  DS_CLK         = 10                       ' DS1620 clock
const  DS_DATA               = 11                ' DS1620 data out

'-------------------- uM-FPU register definitions -----------------------------

const  DegC           = 1                        ' degrees Celsius
const  DegF           = 2                        ' degrees Fahrenheit
const  F1_8           = 3                        ' constant 1.8
const  F32                   = 4                 ' constant 32.0

'-------------------- variables ----------------------------------------------

dim    rawTemp        as integer                 ' raw temperature reading

'=============================================================================
'-------------------- initialization -----------------------------------------
'=============================================================================

setup:
       serout Terminal, Sbaud, Smode, Sint, [13, 10, 13, 10, "Demo 1: "]

       gosub fpu_reset                           ' reset the uM-FPU

       if dataByte = SyncChar then               ' check for synchronization
              gosub print_version                ' print the version string
       else
              serout Terminal, Sbaud, Smode, Sint, ["uM-FPU not detected"]
              goto done
       end if

       gosub init_DS1620                         ' initalize the DS1620
```

```
        'load floating point constants
        '------------------------------
        dataByte = F1_8                         ' select F1_8 as A register
        gosub fpu_sendOp
        dataByte = ATOF                         ' load ASCII string to uM-FPU,
        gosub fpu_sendOp                        '  convert to floating point,
        dataByte = "1"                          '  store in register 0
        gosub fpu_sendByte                      '  select register 0 as B register
        dataByte = "."
        gosub fpu_sendByte
        dataByte = "8"
        gosub fpu_sendByte
        dataByte = 0
        gosub fpu_sendByte
        dataByte = FSET                         ' F1_8 = register 0 (1.8)
        gosub fpu_sendOp

        dataByte = F32                          ' select F32 as A register
        gosub fpu_sendOp
        dataByte = LOADBYTE                     ' load the value 32 to register 0,
        gosub fpu_sendOp                        '  convert to floating point,
        dataByte = 32                           '  select register 0 as the B register
        gosub fpu_sendByte
        dataByte = FSET                         ' F32 = register 0 (32.0)
        gosub fpu_sendOp

'==============================================================================
'-------------------- main routine --------------------------------------------
'==============================================================================

main:
        ' get temperature reading from DS1620
        '------------------------------------
        gosub read_DS1620                       ' get temperature reading from DS1620
        serout Terminal, Sbaud, Smode, Sint, [13,10, 13,10, "Raw Temp:", dec(rawTemp)]

        ' send rawTemp to uM-FPU, convert to floating point, store in register
        '---------------------------------------------------------------------
        dataByte = DegC                         ' select DegC as A register
        gosub fpu_sendOp
        dataByte = LOADWORD                     ' load rawTemp to register 0,
        gosub fpu_sendOp                        '  convert to floating point,
        dataByte = rawTemp.H                    '  select register 0 as B register
        gosub fpu_sendByte
        dataByte = rawTemp.L
        gosub fpu_sendByte
        dataByte = FSET                         ' DegC = register 0 (i.e. set to the
        gosub fpu_sendOp                        '  floatiing point value of razwTemp)

        ' divide the rawTemp by 2 to get degrees Celsius
        '-----------------------------------------------
        dataByte = LOADBYTE                     ' load the value 2 to register 0,
        gosub fpu_sendOp                        '  convert to floating point,
        dataByte = 2                            '  select register 0 as the B register
        gosub fpu_sendByte
        dataByte = FDIV                         ' divide DegC by register 0
        gosub fpu_sendOp                        '  (i.e.. divide by 2)

        ' degF = degC * 1.8 + 32
        '------------------------
        dataByte = degF                         ' select DegF as A register
        gosub  fpu_sendOp
        dataByte = FSET+DegC                    ' set DegF = DegC
        gosub fpu_sendOp
```

```
        dataByte = FMUL+F1_8                    ' multiply DegF by 1.8
        gosub fpu_sendOp
        dataByte = FADD+F32                     ' add 32.0 to DegF
        gosub fpu_sendOp

        'display degrees Celsius
        '-----------------------
        serout Terminal, Sbaud, Smode, Sint, [13, 10, "Degrees C: "]
        dataByte = degC                         ' select DegC as A register
        gosub  fpu_sendOp
        format = 51                             ' print DegC in 5.1 format
        gosub print_floatFormat

        'display degrees Fahrenheit
        '--------------------------
        serout Terminal, Sbaud, Smode, Sint, [13, 10, "Degrees F: "]
        dataByte = degF                         ' select DegF as A register
        gosub  fpu_sendOp
        format = 51                             ' print DegF in 5.1 format
        gosub print_floatFormat

        delay 2000                              ' delay for 2 seconds
        goto main                               ' get the next reading

done:                                           ' end of program
endLoop:                                        ' loop forever at end of program
        goto endloop

'-------------------- init_DS1620 ----------------------------------------------

init_DS1620:
        out DS_RST, 0                           ' initialize pin states
        out DS_CLK, 1
        delay 100

        out DS_RST, 1                           ' configure for CPU control
        shiftout DS_CLK, DS_DATA, 0, &h0C, 8
        shiftout DS_CLK, DS_DATA, 0, &h02, 8
        out DS_RST, 0
        delay 100

        out DS_RST, 1                           ' start temperature conversions
        shiftout DS_CLK, DS_DATA, 0, &hEE, 8
        out DS_RST, 0
        delay 1000                              ' wait for first conversion
        return

'-------------------- read_DS1620 ----------------------------------------------

read_DS1620:
        out DS_RST, 1                           ' read temperature value
        shiftout DS_CLK, DS_DATA, 0, &hAA, 8
        rawTemp = shiftin(DS_CLK, DS_DATA, 0, 9)
        out DS_RST, 0
        return
```

Appendix A
uM-FPU V2 Instruction Summary

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			Write register and select A
FWRITEB	Float	3x	yyyy zzzz		x	Write register and select B
FREAD	Float	4x		yyyy zzzz		Read register
FSET/LSET	Either	5x				A = B
FADD	Float	6x			x	A = A + B
FSUB	Float	7x			x	A = A - B
FMUL	Float	8x			x	A = A * B
FDIV	Float	9x			x	A = A / B
LADD	Long	Ax			x	A = A + B
LSUB	Long	Bx			x	A = A -B
LMUL	Long	Cx			x	A = A * B
LDIV	Long	Dx			x	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 A
FTOA	Float	FA	ff			Convert float to ASCII Store in string buffer
ATOL	Long	FB	aa … 00		0	Convert ASCII to long Store in A
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		bb		Get lower 16 bits of register A
READLONG	Long	FE92		bb		Get long integer value of register A
READFLOAT	Float	FE93		bb		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	A = A / B (unsigned long) Remainder stored in register 0
POWER	Float	FEE0				A = A ** 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				Right parenthesis
RIGHT		FEEF			0	Left 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 is approximately ± $10^{38.53}$.

.

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 hex data constants are as follows:

```
&h00, &h00, &h00, &h00     '0.0
&h3D, &hCC, &hCC, &hCD     '0.1
&h3F, &h00, &h00, &h00     '0.5
&h3F, &h40, &h00, &h00     '0.75
&h3F, &h7F, &hF9, &h72     '0.9999
&h3F, &h80, &h00, &h00     '1.0
&h40, &h00, &h00, &h00     '2.0
&h40, &h2D, &hF8, &h54     '2.7182818 (e)
&h40, &h49, &h0F, &hDB     '3.1415927 (pi)
&h41, &h20, &h00, &h00     '10.0
&h42, &hC8, &h00, &h00     '100.0
&h44, &h7A, &h00, &h00     '1000.0
&h44, &h9A, &h52, &h2B     '1234.5678
&h49, &h74, &h24, &h00     '1000000.0
&h80, &h00, &h00, &h00     '-0.0
&hBF, &h80, &h00, &h00     '-1.0
&hC1, &h20, &h00, &h00     '-10.0
&hC2, &hC8, &h00, &h00     '-100.0
&h7F, &hC0, &h00, &h00     'NaN (Not-a-Number)
&h7F, &h80, &h00, &h00     '+inf
&hFF, &h80, &h00, &h00     '-inf
```