Micromega Corporation

Using uM-FPU V2 with the PICmicro® Microcontroller

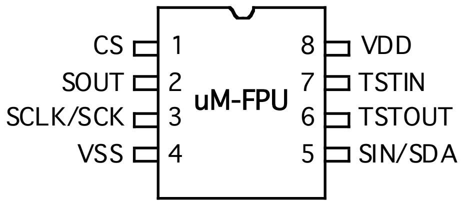
Introduction

The uM-FPU is a 32-bit floating point coprocessor that is easily interfaced with the with the Microchip PICmicro® family of microcontrollers to provide support for 32-bit floating point 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. 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

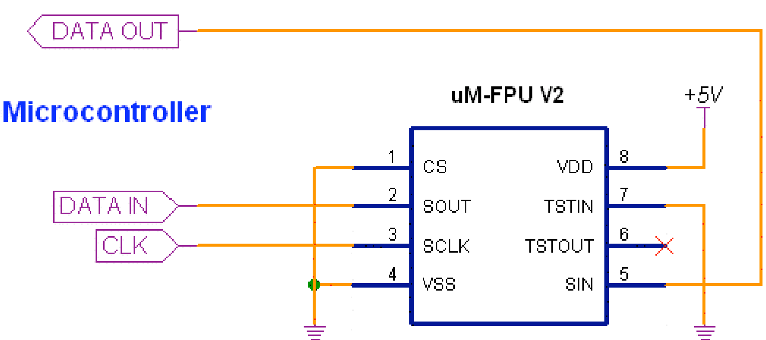
Connecting the uM-FPU to the Microchip PICmicro® using SPI

The uM-FPU can be connected using either a 2-wire or 3-wire. The 2-wire connection uses a clock signal and a bidirectional data signal and requires the program to change the input/output direction of the pin as required. The 3-wire connection uses a clock signal and separate data input and data output signals. The support routines assume a 3-wire SPI interface. The default settings for these pins are:

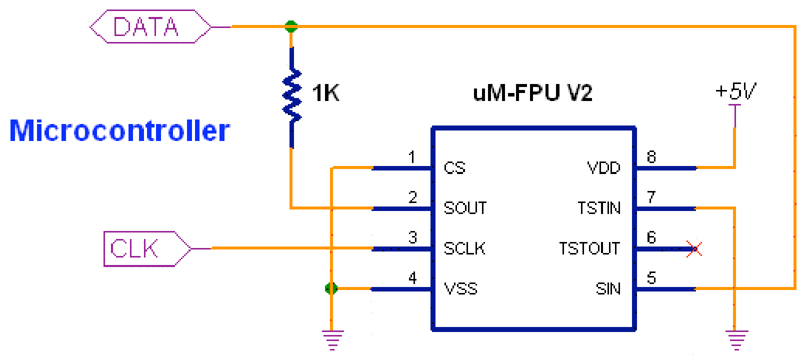
Pin	PIC16F877	PIC16F628
`FPU_CLOCK`	`RC3`	`RB7`
`FPU_DATAIN`	`RC4`	`RB6`
`FPU_DATAOUT`	`RC5`	`RB5`

The settings of these pins can be changed to suit your application. The default settings for the PIC16F877 allow the hardware SPI support to be used. By default, the uM-FPU chip is always selected, so the FPU_CLOCK and FPU_DATAIN/FPU_DATAOUT pins should not be used for other connections as this will likely result in loss of synchronization between the PICmicro and the uM-FPU coprocessor.

3-wire SPI Connection



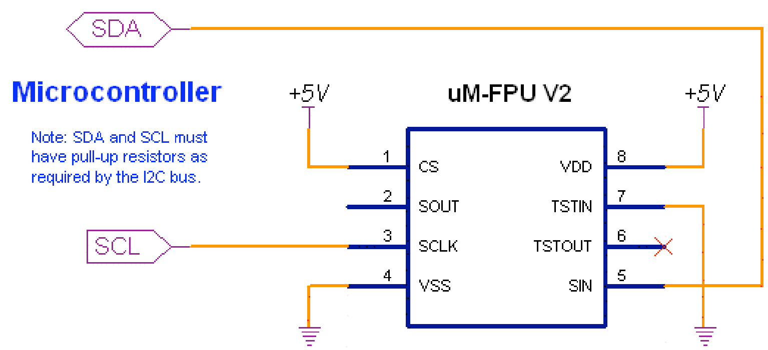
2-wire SPI Connection



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 PICmicro using I²C

The uM-FPU V2 can also be connected using an I²C 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 I²C interface.



An Introduction to the uM-FPU

The following section provides an introduction to the uM-FPU using PICmicro MPASM assembler code for all 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 W register with the byte value to send and calling the `fpu_readByte` routine. For example:

```
movlw  FADD+5
call   fpu_sendByte
```

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:

```
movlw  SQRT
call   fpu_sendByte
```

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 $FE, defined as XOP. To use an extended opcode, you send the XOP byte first, followed by the extended opcode. For example:

```
movlw  XOP
call   fpu_sendByte
movlw  ATAN
call   fpu_sendByte
```

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 instruction selects register 0 as the B register then calculates A = A + B.

```
movlw    FADD
call     fpu_sendByte
```

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

```
movlw  FADD+5
call   fpu_sendByte
```

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 instruction to perform that operation is as follows:

```
movlw    SELECTA+2            ;select register 2 as the A register
call     fpu_sendByte
movlw    FADD+5               ;select register 5 as the B register
call     fpu_sendByte         ; 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:

```
#define  Total 2              ;total amount  (uM-FPU register)
#define  Count 5              ;current count (uM-FPU register)

movlw    SELECTA+Total        ;select register Total as the A register
call     fpu_sendByte
movlw    FADD+Count           ;select register Count as the B register
call     fpu_sendByte         ; and calculate A = A + B
```

Selecting the A register is such a common occurrence, it was defined as opcode $0x. The definition for SELECTA is 0x00, so SELECTA+Total is the same as just using Total by itself. Using this shortcut, the same example would now be:

```
movlw    Total                ;select register Total as the A register
call     fpu_sendByte
movlw    FADD+Count           ;select register Count as the B register
call     fpu_sendByte         ; and calculate A = A + B
```

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 PICmicro 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 two byte (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 instruction is used:

```
movlw   DegC                ;select DegC as A register
call    fpu_sendByte

movlw   LOADWORD            ;load rawTemp to register 0,
call    fpu_sendByte        ; convert to floating point,
movf    rawTemp+1, w        ; select register 0 as B register
call    fpu_sendByte
movf    rawTemp, w
call    fpu_sendByte

movlw   FSET                ;degC = register 0 (i.e. set to the
call    fpu_sendByte        ; floating point value of rawTemp)
```

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

```
movlw   LOADBYTE            ;load the value 2 to register 0,
call    fpu_sendByte        ; convert to floating point,
movlw   .2                  ; select register 0 as B register
call    fpu_sendByte

movlw   FDIV                ;divide DegC by register 0
call    fpu_sendByte
```

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:

```
movlw   F1_8                ;select F1_8 as A register
call    fpu_sendByte

movlw   ATOF                ;load the string 1.8 to the uM-FPU
call    fpu_sendByte        ; (note: string must be zero terminated)
movlw   '1'                 ; convert to floating point,
call    fpu_sendByte        ; store value in register 0
movlw   '.'                 ; select register 0 as the B register
call    fpu_sendByte
movlw   '8'
call    fpu_sendByte
movlw   0
call    fpu_sendByte

movlw   FSET                ;F1_8 = register 0 (i.e. 1.8)
call    fpu_sendByte
```

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

```
movlw   F32                 ;select F32 as A register
call    fpu_sendByte
movlw   LOADBYTE            ;load the byte value 32 to register 0,
```

```
call     fpu_sendByte          ; convert to floating point,
movlw    .32                   ; select register 0 as B register
call     fpu_sendByte

movlw    FSET                  ;F32 = register 0 (i.e. 32.0)
call     fpu_sendByte
```

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

```
movlw    DegF                  ;select DegF as the A register
call     fpu_sendByte

movlw    FSET+DegC             ;set DegF = DegC
call     fpu_sendByte

movlw    FMUL+F1_8             ;multiply DegF by 1.8
call     fpu_sendByte

movlw    FADD+F32              ;add 32.0 to DegF
call     fpu_sendByte
```

Now we print the results. There are support routines provided for printing floating point numbers. The print_float routine prints an unformatted floating point value and displays up to eight digits of precision. The print_floatFormat routine prints a formatted floating point number. We'll use print_floatFormat to display the results. The desired format is loaded into the W register. 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.

```
movlw    DegC                  ;select DegC as A register
call     fpu_sendByte

movlw    .51                   ;set format to 5,1
call     print_floatFormat     ;print floating point value
```

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

A full set of assembler support routines is provided to handle all of the communication between the PICmicro and the uM-FPU. The routines are designed for use with the MPLAB IDE using the MPASM Assembler and MPLINK Object Linker. The routines could easily be adapted to other assemblers. The interface files are as follows:

umfpu.asm	High level routines for each uM-FPU function
umfpu.inc	Include file containing definitions for each uM-FPU instruction opcode
fpusw_4.asm	Software SPI interface routine (bit-bang), 4 MHz
fpusw_20.asm	Software SPI interface routine (bit-bang), 20 MHz
fpuhw_4.asm	Hardware SPI interface routine, 4 MHz
fpuhw_20.asm	Hardware SPI interface routine, 20 MHz
delay_4.asm	Delay routine, 4 Mhz
delay_20.asm	Delay routine, 20 Mhz
serial.asm	Serial port routines to print data

MPLAB project files and linker files are provided for each of the sample applications. The files can be used directly to test the sample applications, or used as the starting point for a new program. Each uM-FPU support routine is described below.

The following routines are provided in the files *fpusw_xx* and *fpuhw_xx*.

fpu_reset
To ensure that the PICmicro 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 sets the Z flag to 1 if successful, or 0 if the reset failed.

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 using the fpu_readByte routine, a call to fpu_wait must be made prior to sending the read instruction. An example of reading a byte value is as follows (the fpu_readDelay routine is described later):

```
call     fpu_wait              ;wait for uM-FPU to be ready
movlw    XOP                   ;read byte of data
call     fpu_sendByte
movlw    READBYTE
call     fpu_sendByte

call     fpu_readDelay         ;wait for read setup delay

call     fpu_readByte          ;read the byte
```

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.

fpu_sendByte
Sends an 8-bit value to the uM-FPU. This routine is used for sending all instructions and data. The byte to send is loaded in the W register before calling the routine.

fpu_readByte
Reads an 8-bit value from the uM-FPU. The uM-FPU must have received a read instruction and be ready to send data before this routine is called. The byte read from the uM-FPU is returned in the W register. The Z flag is also set according to the value of the W register.

fpu_readDelay
After a read instruction is sent, and before the first fpu_readByte call, a setup delay is required to ensure that the uM-FPU is ready to send data. The fpu_readDelay routine provides the required read setup delay. For read instructions that return multiple bytes, the fpu_readDelay call is only required before the first byte.

The following routines are provide in the file *serial.asm*.

print_setup
Initializes the serial port. This routine must be called in the initialization section of the program.

print_version
Prints the uM-FPU version string to the serial port.

print_float
The value in register A is sent to the serial port as a floating point value. 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 sent to the serial port as a formatted floating point value. The desired format is loaded into the W register. 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 sent to the serial port as a signed long integer. 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 sent to the serial port as a formatted long integer. The desired format is loaded into the W register. 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

print_string
A zero terminated string is sent to the serial port. The lower part of the string address is loaded in the W register before calling print_string. Strings are stored in a special data area called STRINGS that is defined in the linker file.

print_byte
Sends the 8-bit value in the W register to the serial port.

print_hex
Sends the 8-bit value in the W register to the serial port as two hexadecimal digits.

print_hexDigit
Sends the lower 4-bits of the W register to the serial port as a hexadecimal digit.

print_crlf
Sends a carriage return and linefeed to the serial port.

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

```
movlw   Result              ;select Result as the A register
call    fpu_sendByte

movlw   LOADBYTE            ;load the byte value 20 to register 0,
call    fpu_sendByte        ; convert to floating point,
movlw   .20                 ; select register 0 as B register
call    fpu_sendByte

movlw   FSET                ;Result = register 0 (i.e. 20.0)
call    fpu_sendByte
```

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

```
movlw   Total               ;select Total as the A register
call    fpu_sendByte

movlw   XOP                 ;load the byte value 100 to register 0,
call    fpu_sendByte        ; convert to floating point,
movlw   LONGBYTE            ; select register 0 as B register
call    fpu_sendByte
movlw   .100
call    fpu_sendByte

movlw   LDIV                ;divide Total by register 0
call    fpu_sendByte        ; (i.e. divide by 100)
```

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

```
movlw   Result              ;select Result as the A register
call    fpu_sendByte

movlw   XOP                 ;load 0.0 to register 0,
call    fpu_sendByte        ; select register 0 as B register
movlw   LOADZERO
```

```
    call     fpu_sendByte

    movlw    FSET                  ;set Result to the value in register 0
    call     fpu_sendByte          ; (i.e. 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 0x41A00000)

```
    movlw    WRITEA+Angle          ;select Angle as the A register,
    call     fpu_sendByte

    movlw    0x41                  ;load 0x41A00000 to A register
    call     fpu_sendByte          ;(32-bit floating point value 20.0)
    movlw    0xA0
    call     fpu_sendByte
    movlw    0x00
    call     fpu_sendByte
    movlw    0x00
    call     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

```
    movlw    XOP                   ;select Total as the A register
    call     fpu_sendByte
    movlw    LWRITEA+Angle
    call     fpu_sendByte

    movlw    0x00                  ;load 0x0007A120 to A register
    call     fpu_sendByte          ;(32-bit long integer value 500000)
    movlw    0x07
    call     fpu_sendByte
    movlw    0xA1
    call     fpu_sendByte
    movlw    0x20
    call     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

```
    movlw    Angle                 ;select Angle as A register
    call     fpu_sendByte

    movlw    ATOF                  ;load the string 1.5885 to the uM-FPU
    call     fpu_sendByte          ; convert to floating point,
    movlw    '1'                   ; store value in register 0
    call     fpu_sendByte          ; select register 0 as the B register
    movlw    '.'
    call     fpu_sendByte
    movlw    '5'
```

```
call     fpu_sendByte
movlw    '8'
call     fpu_sendByte
movlw    '8'
call     fpu_sendByte
movlw    '5'
call     fpu_sendByte
movlw    0                      ;note: string must be zero terminated
call     fpu_sendByte

movlw    FSET                   ;F1_8 = register 0 (i.e. 1.8)
call     fpu_sendByte
```

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 instructions read the floating point value from the A register

```
call fpu_wait                   ;wait for uM-FPU to be ready

movlw    XOP                    ;read floating point value A register
call     fpu_sendByte
movlw    READFLOAT
call     fpu_sendByte

call     fpu_readDelay          ;wait for read setup delay

call     fpu_readByte           ;read 32-bit value as four bytes
movwf    fval+3                 ;(most significant byte first)
call     fpu_readByte
movwf    fval+2
call     fpu_readByte
movwf    fval+1
call     fpu_readByte
movwf    fval
```

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 instructions are used to read a byte value from the lower 8 bits of A register

```
call     fpu_wait               ;wait for uM-FPU to be ready

movlw    XOP                    ;read byte of data
call     fpu_sendByte
movlw    READBYTE
call     fpu_sendByte

call     fpu_readDelay          ;wait for read setup delay

call     fpu_readByte           ;read the byte
```

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 stored in the `status` variable. The bit definitions for the `status` variable are as follows:

bit 0	Zero bit	(0 – not zero, 1 – zero)
bit 1	Sign bit	(0 – positive, 1 – negative)
bit 2	Not-a-Number	(0 – valid number, 1 – NaN)
bit 3	Infinity	(0 – not infinite, 1 – infinite)

For example:

```
    #define  ZERO 0                ;zero status bit
    #define  SIGN 1                ;sign status bit
    #define  NAN  2                ;Not-a-Number status bit
    #define  INF  3                ;Infinity status bit

    call     fpu_wait              ;wait for uM-FPU to be ready

    movlw    FSTATUS               ;send FSTATUS instruction
    call     fpu_sendByte

    call     fpu_readDelay         ;wait for read setup delay

    call     fpu_readByte          ;read status byte and store
    movwf    status

    btfsc    status, ZERO          ;check status bits
    goto     zeroValue
    btfsc    status, SIGN
    goto     negativeValue

                                   ;value is positive
    ...
  negativeValue
                                   ;value is negative
    ...
  zeroValue
                                   ;value is zero
    ...
```

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:

```
    call     fpu_wait              ;wait for uM-FPU to be ready

    movlw    FCOMPARE              ; send FCOMPARE instruction
    call     fpu_sendByte

    call     fpu_readDelay         ;wait for read setup delay

    call     fpu_readByte          ;read status byte and store
    movwf    status

    btfsc    status, ZERO          ;check the status bits
    goto     sameAs
    btfsc    status, SIGN
    goto     lessThan

    ...                            ;A > B
```

```
   lessThan
                                 ;A < B"
   ...
   sameAs
                                 ;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. A bit definition is provided for each status bit in the `status` variable. They are as follows:

bit 0	Zero bit	(0 – not zero, 1 – zero)
bit 1	Sign bit	(0 – positive, 1 – negative)

For example:

```
   #define  ZERO 0               ;zero status bit
   #define  SIGN 1               ;sign status bit

   call     fpu_wait             ;wait for uM-FPU to be ready

   movlw    XOP                  ;send LSTATUS instruction
   call     fpu_sendByte
   movlw    LSTATUS
   call     fpu_sendByte


   call     fpu_readDelay        ;wait for read setup delay

   call     fpu_readByte         ;read status byte and store
   movwf    status

   btfsc    status, ZERO         ;check status bits
   goto     zeroValue
   btfsc    status, SIGN
   goto     negativeValue

                                 ;value is positive
   ...
  negativeValue
                                 ;value is negative
   ...
  zeroValue
                                 ;value is zero
   ...
```

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 instructions compare the values in registers Value1 and Value2.

```
   call     fpu_wait             ;wait for uM-FPU to be ready

   movlw    Value1               ;select Value1 as A register
   call     fpu_sendByte

   movlw    SELECTB+Value2       ;select Value2 as B register
   call     fpu_sendByte

   movlw    XOP                  ;send LCOMPARE instruction
   call     fpu_sendByte
   movlw    LCOMPARE
```

```
    call     fpu_sendByte


    call     fpu_readDelay          ;wait for read setup delay

    call     fpu_readByte           ;read status byte and store
    movwf    status

    btfsc    status, ZERO           ;check the status bits
    goto     sameAs
    btfsc    status, SIGN
    goto     lessThan

    ...                             ;A > B

    lessThan
                                    ;A < B"
    ...
    sameAs
                                    ;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.

```
#define Xvalue      1      ;X value (uM-FPU register 1)
#define Yvalue      2      ;Y value (uM-FPU register 2)
#define Zvalue      3      ;Z value (uM-FPU register 3)

    movlw    Zvalue                 ;select Zvalue as the A register
    call     fpu_sendByte

    movlw    FSET+Xvalue            ;Zvalue = Xvalue
    call     fpu_sendByte

    movlw    FMUL+Xvalue            ;Zvalue = Zvalue * Xvalue (i.e. X**2)
    call     fpu_sendByte

    movlw    XOP                    ;save current A register selection,
    call     fpu_sendByte           ; select temporary register as A register (temp)
    movlw    LEFT
    call     fpu_sendByte

    movlw    FSET+Yvalue            ;temp = Yvalue
    call     fpu_sendByte

    movlw    FMUL+Yvalue            ;temp = temp * Yvalue (i.e. Y**2)
    call     fpu_sendByte
```

```
    movlw    XOP                    ;store temp to register 0,
    call     fpu_sendByte           ; select Zvalue as A
    movlw    RIGHT                  ; (previously saved selection)
    call     fpu_sendByte

    movlw    FADD                   ;add register 0 to Zvalue (i.e. X**2 + Y**2)
    call     fpu_sendByte

    movlw    SQRT                   ;take the square root of Zvalue
    call     fpu_sendByte
```

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

```
    movlw    Yvalue                 ;select Yvalue as the A register
    call     fpu_sendByte

    movlw    LOADBYTE               ;load the value 10 to register 0,
    call     fpu_sendByte           ; convert to floating point,
    movlw    .10                    ; select register 0 as the B register
    call     fpu_sendByte

    movlw    FSET                   ;Yvalue = 10.0
    call     fpu_sendByte

    movlw    XOP                    ;save current A register selection
    call     fpu_sendByte           ; select temporary register as A register (temp)
    movlw    LEFT
    call     fpu_sendByte

    movlw    FSET+Xvalue            ;temp = Xvalue
    call     fpu_sendByte

    movlw    XOP                    ;load 1.0 to register 0,
    call     fpu_sendByte           ; select register 0 as the B register
    movlw    LOADONE
    call     fpu_sendByte

    movlw    FADD                   ; temp = temp + 1 (i.e. X+1)
    call     fpu_sendByte

    movlw    XOP                    ;store temp to register 0,
    call     fpu_sendByte           ; select Yvalue as A
    movlw    RIGHT                  ; (previously saved selection)
    call     fpu_sendByte

    movlw    FDIV                   ; divide Yvalue by the value in register 0
    call     fpu_sendByte
```

Alternate Floating Point Format

Several compilers for the PICmicro® use a slightly modified version of the standard IEEE 754 floating point format. The alternate format is shown below:

Exponent	S	Mantissa
31 24	23	22 0

The uM-FPU uses the standard IEEE 754 format (as described in Appendix B) by default, but it can also support the alternate PIC format. To use the PIC floating point format, the following function call should be made immediately after a reset:

```
movlw    PICMODE
call     fpu_sendByte
```

All internal data on the uM-FPU is still stored in IEEE 754 format, but when the uM-FPU is in PIC mode an automatic conversion is done by the `FREAD, FWRITEA, FWRITEB, and READFLOAT` instructions so the PIC program use floating point data in the alternate format. The mode parameter bytes stored in Flash memory can also be set with the debug monitor so that PIC floating point format is automatically selected at reset (see the uM-FPU Datasheet). The IEEEMODE instruction can be used to switch back to standard IEEE 754 floating point mode.

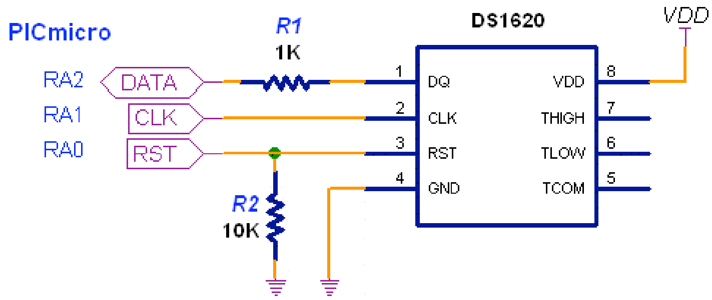
Further Information

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.asm)

```
;This program demonstrates how to use the uM-FPU floating point coprocessor
;connected to PIC microcontroller over a 3-wire 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.

   list      p=16f877                    ;list directive to define processor
   #include <p16f877.inc>                ;processor specific variable definitions

   __CONFIG _CP_OFF & _WDT_OFF & _BODEN_ON & _PWRTE_ON & _HS_OSC
       & _WRT_ENABLE_ON & _LVP_OFF & _DEBUG_OFF & _CPD_OFF

   #include umfpu.inc                    ;uM-FPU definitions

   extern    delay_ms
   extern    print_setup, print_byte, print_crlf, print_string
   extern    print_version, print_floatFormat, print_hex

;-------------------- uM-FPU register definitions -----------------------------

#define    DegC              1             ;degrees Celsius
#define    DegF              2             ;degrees Fahrenheit
#define    F1_8              3             ;constant 1.8
#define    F32               4             ;constant 32.0

;-------------------- variable definitions ------------------------------------
           udata

#define    DS_RST            PORTA, 0      ;DS1620 reset/enable
#define    DS_CLK            PORTA, 1      ;DS1620 clock
#define    DS_DATA           PORTA, 2      ;DS1620 data
#define    DS_DATA_DIR       TRISA, 2      ;direction bit

dataBits   res    1                        ;data bits
bitCount   res    1                        ;bit count
delayCnt   res    1                        ;counter for main loop delay
rawTemp    res    2                        ;raw temperature

;-------------------- string definitions --------------------------------------
; lower byte of stringTable address must be 00
; stringTable can have up to 255 bytes of data
; all strings must be zero terminated

STRINGS       code
       global stringTable
```

```
stringTable
   addwf     PCL,f                       ;computed goto for strings

start1Message
       dt     0x0D, 0x0A
       dt     0x0D, 0x0A, "Demo1 - ", 0
start2Message
       dt     0x0D, 0x0A, "-------------------", 0x0D, 0x0A, 0
errorMessage
       dt     "uM-FPU not detected", 0
rawString
       dt 0x0D, 0x0A, "Raw Temp: ", 0
degCString
       dt ", Degrees C:", 0
degFString
       dt ", Degrees F:", 0

;-------------------- reset and interrupt vector -----------------------------

STARTUP       code
   nop        ;reset vector
   goto    reset
   nop
   nop
   goto      isr                         ;interrupt vector

;-------------------- interrupt service routine ------------------------------

PROG          code
isr
   retfie    ;(no interrupts used)


;==============================================================================
;-------------------- initialization ------------------------------------------
;==============================================================================

reset
   call      print_setup                 ;initialize the serial port

   movlw     LOW start1Message           ;display startup string
   call      print_string

   ; reset the uM-FPU and display version
   ;-------------------------------------

   call      fpu_reset                   ;reset the uM-FPU
   btfsc     STATUS, Z                   ;check status
   goto      reset2
   movlw     LOW errorMessage            ;print error message if reset failed
   call      print_string
   goto      done

reset2
   call      print_version               ;print uM-FPU version number
   movlw     LOW start2Message           ;display underline
   call      print_string

   call      init_DS1620                 ;initizlize the DS1620

   ; load constant 1.8
   ;------------------
   movlw     F1_8                        ;select F1_8 as A register
   call      fpu_sendByte
   movlw     ATOF                        ;load the string 1.8 to the uM-FPU
```

```
    call      fpu_sendByte               ; (note: string must be zero terminated)
    movlw     '1'                        ; convert to floating point,
    call      fpu_sendByte               ; store value in register 0
    movlw     '.'                        ; select register 0 as the B register
    call      fpu_sendByte
    movlw     '8'
    call      fpu_sendByte
    movlw     0
    call      fpu_sendByte
    movlw     FSET                       ;F1_8 = register 0 (i.e. 1.8)
    call      fpu_sendByte

    ; load constant 32.0
    ;-------------------
    movlw     F32                        ;select F32 as A register
    call      fpu_sendByte
    movlw     LOADBYTE                   ;load the byte value 32 to register 0,
    call      fpu_sendByte               ; convert to floating point,
    movlw     .32                        ; select register 0 as B register
    call      fpu_sendByte
    movlw     FSET                       ;F32 = register 0 (i.e. 32.0)
    call      fpu_sendByte

;==============================================================================
;-------------------- main routine --------------------------------------------
;==============================================================================

main
    ;get temperature reading from DS1620
    ;-----------------------------------
    call      read_temperature           ;read temperature

    movlw     LOW rawString              ;display string
    call      print_string

    movf      rawTemp+1, w               ;display raw Temperature as hex
    call      print_hex
    movf      rawTemp, w
    call      print_hex

    ;load rawTemp to uM-FPU, convert to floating point, and store in register
    ;--------------------------------------------------------------------------
    movlw     DegC                       ;select DegC as A register
    call      fpu_sendByte
    movlw     LOADWORD                   ;load rawTemp to register 0,
    call      fpu_sendByte               ; convert to floating point,
    movf      rawTemp+1, w               ; select register 0 as B register
    call      fpu_sendByte
    movf      rawTemp, w
    call      fpu_sendByte
    movlw     FSET                       ;degC = register 0 (i.e. set to the
    call      fpu_sendByte               ; floating point value of rawTemp)

    ;divide the raw value by 2 to get degrees Celsius
    ;-------------------------------------------------
    movlw     LOADBYTE                   ;load the value 2 to register 0,
    call      fpu_sendByte               ; convert to floating point,
    movlw     .2                         ; select register 0 as B register
    call      fpu_sendByte
    movlw     FDIV                       ;divide DegC by register 0
    call      fpu_sendByte               ; (i.e. divide by 2)

    ;DegF = DegC * 1.8 + 32
    ;----------------------
    movlw     DegF                       ;select DegF as the A register
```

```
   call      fpu_sendByte
   movlw     FSET+DegC                   ;set DegF = DegC
   call      fpu_sendByte
   movlw     FMUL+F1_8                   ;multiply DegF by 1.8
   call      fpu_sendByte
   movlw     FADD+F32                    ;add 32.0 to DegF
   call      fpu_sendByte

   ;display degrees Celsius
   ;-----------------------
   movlw     LOW degCString              ;display text string
   call      print_string

   movlw     DegC                        ;select DegC as A register
   call      fpu_sendByte

   movlw     .51                         ;set format to 5,1
   call      print_floatFormat           ;print floating point value

   ;display degrees Fahrenheit
   ;--------------------------
   movlw     LOW degFString              ;display text string
   call      print_string

   movlw     DegF                        ;select DegF as A register
   call      fpu_sendByte

   movlw     .51                         ;set format to 5,1
   call      print_floatFormat           ;print floating point value

   ;delay for 2 seconds and repeat main loop
   ;-----------------------------------------
   movlw     8                           ;8 x 250 msec = 2 seconds
   movwf     delayCnt

pause
   movlw     .250                        ;delay for 1 second
   call      delay_ms
   decfsz    delayCnt, f                 ;loop for all bits
   goto      pause

   goto      main                        ;repeat main loop

;-------------------- init_DS1620 --------------------------------------------
; initialize DS1620
;-----------------------------------------------------------------------------

init_DS1620
   bcf       DS_RST                      ;initialize pins
   bsf       DS_CLK
   bankselTRISA
   movlw     0x06                        ;congiure A0-A3 for digitial I/O
   movwf     ADCON1
   movlw     0xF8                        ;configure A0-A2 as outputs
   movwf     TRISA
   banksel   PORTA

   movlw     .100                        ;delay
   call      delay_ms

   bsf       DS_RST                      ;configure for CPU control
   movlw     0x0C
   call      write_DS1620
   movlw     0x02
   call      write_DS1620
```

```
   bcf       DS_RST

   movlw     .100                       ;delay
   call      delay_ms

   bsf       DS_RST                     ;start temperature conversions
   movlw     0xEE
   call      write_DS1620
   bcf       DS_RST

   movlw     .250                       ;delay 1 second
   call      delay_ms
   movlw     .250
   call      delay_ms
   movlw     .250
   call      delay_ms
   movlw     .250
   call      delay_ms
   return

done
   goto      done                       ;error exit

;-------------------- write_DS1620 -------------------------------------------
; write DS1620 command
;-----------------------------------------------------------------------------

write_DS1620
   movwf     dataBits                   ;save the byte to send
   movlw     .8                         ;get number of bits to send
   movwf     bitCount

write2
   bcf       DS_DATA                    ;set data output LOW
   rrf       dataBits, f                ;get next data bit
   btfsc     STATUS, C                  ;if next bit is 1, set data output HIGH
   bsf       DS_DATA

   bcf       DS_CLK                     ;pulse the clock
   bsf       DS_CLK

   decfsz    bitCount, f                ;loop for all bits
   goto      write2
   return

;-------------------- read_temperature ---------------------------------------
; read temperature value from DS1620 and store in rawTemp
;-----------------------------------------------------------------------------

read_temperature
   bsf       DS_RST                     ;enable DS1620

   movlw     0xAA                       ;send read temperature command
   call      write_DS1620

   banksel   TRISA
   bsf       DS_DATA_DIR                ;configure DS_DATA as input
   banksel   PORTA

   movlw     .8                         ;get number of bits to receive
   movwf     bitCount

read2
   bcf       DS_CLK                     ;set clock LOW
   bcf       STATUS, C                  ;set carry to zero
```

```
btfsc     DS_DATA                     ;check data input
bsf       STATUS, C                   ;if HIGH, set carry to one
bsf       DS_CLK                      ;set clock HIGH
rrf       rawTemp, f                  ;store next data bit
decfsz    bitCount, f                 ;loop for all bits
goto      read2

clrw                                  ;clear high bits
btfsc     DS_DATA                     ;check data input
movlw     0xFF                        ;if HIGH, set high bits
bsf       DS_CLK                      ;set clock HIGH
movwf     rawTemp+1                   ;store high bits

bsf       DS_CLK                      ;set clock HIGH
banksel   TRISA
bcf       DS_DATA_DIR                 ;configure DS_DATA as output
banksel   PORTA
bcf                                   DS_RST ;disable DS1620
return

end
```

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 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 four byte values are as follows:

```
0x00, 0x00, 0x00, 0x00     ;0.0
0x3D, 0xCC, 0xCC, 0xCD     ;0.1
0x3F, 0x00, 0x00, 0x00     ;0.5
0x3F, 0x40, 0x00, 0x00     ;0.75
0x3F, 0x7F, 0xF9, 0x72     ;0.9999
0x3F, 0x80, 0x00, 0x00     ;1.0
0x40, 0x00, 0x00, 0x00     ;2.0
0x40, 0x2D, 0xF8, 0x54     ;2.7182818 (e)
0x40, 0x49, 0x0F, 0xDB     ;3.1415927 (pi)
0x41, 0x20, 0x00, 0x00     ;10.0
0x42, 0xC8, 0x00, 0x00     ;100.0
0x44, 0x7A, 0x00, 0x00     ;1000.0
0x44, 0x9A, 0x52, 0x2B     ;1234.5678
0x49, 0x74, 0x24, 0x00     ;1000000.0
0x80, 0x00, 0x00, 0x00     ;-0.0
0xBF, 0x80, 0x00, 0x00     ;-1.0
0xC1, 0x20, 0x00, 0x00     ;-10.0
0xC2, 0xC8, 0x00, 0x00     ;-100.0
0x7F, 0xC0, 0x00, 0x00     ;NaN (Not-a-Number)
0x7F, 0x80, 0x00, 0x00     ;+inf
0xFF, 0x80, 0x00, 0x00     ;-inf
```