Using uM-FPU V3.1 with the PICBASIC PRO Compiler

Micromega Corporation

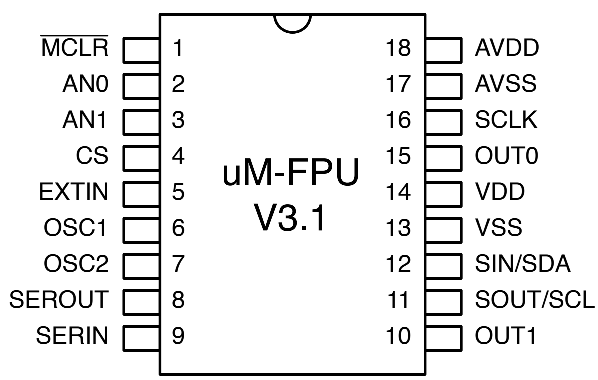
Introduction

The uM-FPU V3.1 chip is a 32-bit floating point coprocessor that can be easily interfaced with Microchip Technology PIC® microcontrollers, and programmed using the PICBASIC PRO™ Compiler from microEngineering Labs, to provide support for 32-bit IEEE 754 floating point operations and 32-bit long integer operations. The uM-FPU V3.1 chip supports both I^{2}C and SPI connections.

This document describes how to use the uM-FPU V3.1 chip with the PICBASIC PRO Compiler. For a full description of the uM-FPU V3.1 chip, please refer to the *uM-FPU V3.1 Datasheet* and *uM-FPU V3.1 Instruction Reference*. Application notes are also available on the Micromega website.

uM-FPU V3.1 Pin Diagram and Pin Description

DIP-18, SOIC-18



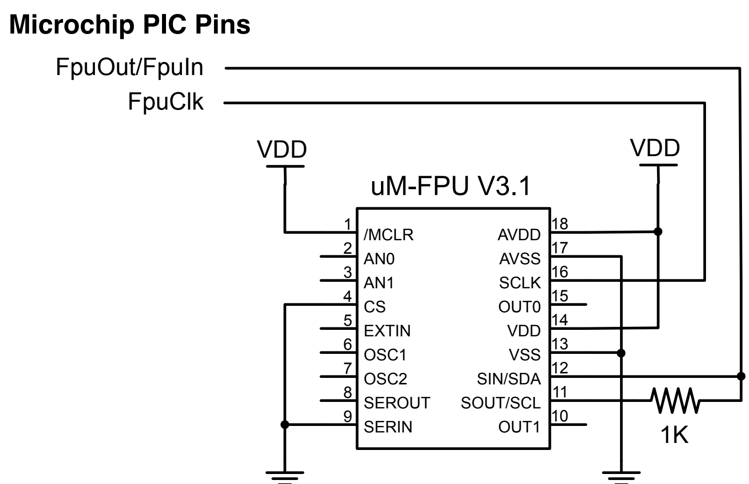
Pin	Name	Type	Description
1	/MCLR	Input	Master Clear (Reset)
2	AN0	Input	Analog Input 0
3	AN1	Input	Analog Input 1
4	CS	Input	Chip Select, Interface Select
5	EXTIN	Input	External Input
6	OSC1	Input	Oscillator Crystal (optional)
7	OSC2	Output	Oscillator Crystal (optional)
8	SEROUT	Output	Serial Output, Debug Monitor - Tx
9	SERIN	Input	Serial Input, Debug Monitor - Rx
10	OUT1	Output	Digital Output 1
11	SOUT SCL	Output Input	SPI Output, Busy/Ready Status I^{2}C Clock

12	SIN SDA	Input In/Out	SPI Input I^2C Data
13	VSS	Power	Digital Ground
14	VDD	Power	Digital Supply Voltage
15	OUT0	Output	Digital Output 0
16	SCLK	Input	SPI Clock
17	AVSS	Power	Analog Ground
18	AVDD	Power	Analog Supply Voltage

Connecting the Microchip PIC using 2-wire SPI

Only two pins are required for interfacing a Microchip PIC to the uM-FPU V3.1 chip using a 2-wire SPI interface. The communication uses a bidirectional serial interface that requires a clock pin and a data pin. An example of the pin settings for a Microchip PIC are shown below. (They can be changed to suit your application.)

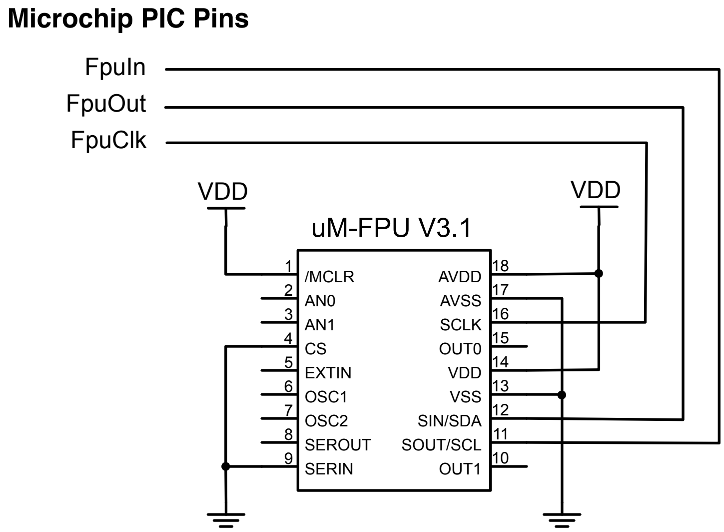
```
FpuClk    var    PORTC.3       ' pin RC3
FpuIn     var    PORTC.4       ' pin RC4
FpuOut    var    PORTC.4       ' pin RC4
```



Connecting the Microchip PIC using 3-wire SPI

Three pins are required for interfacing a Microchip PIC to the uM-FPU V3.1 chip using a 3-wire SPI interface. The communication uses a clock pin, an input data pin, and an output data pin. An example of the pin settings for a Microchip PIC are shown below. (They can be changed to suit your application.)

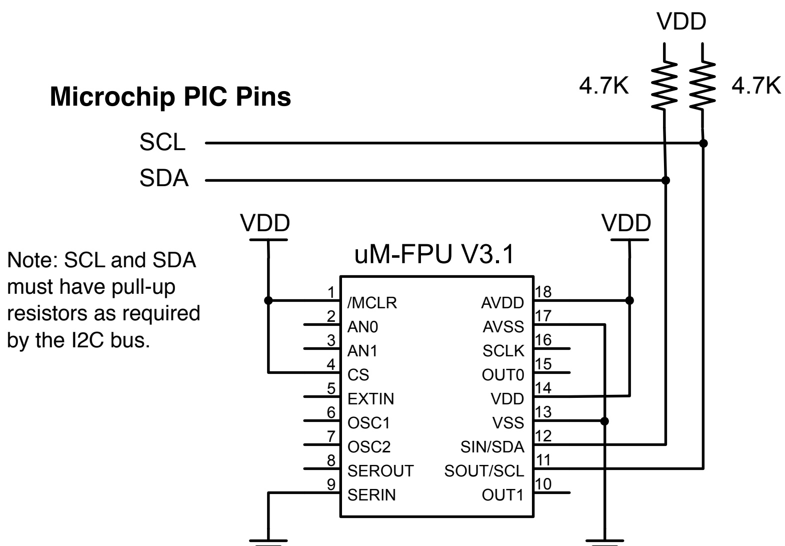
```
FpuClk    var    PORTC.3       ' pin RC3
FpuIn     var    PORTC.4       ' pin RC4
FpuOut    var    PORTC.5       ' pin RC5
```



Connecting the Microchip PIC using I²C

The uM-FPU V3.1 can also be connected using an I²C interface. The default slave ID for the uM-FPU chip is $C8. An example of the pin settings for a Microchip PIC are shown below (they can be changed to suit your application):

```
Fpu_SCL   var   PORTC.3      ' pin RC3
Fpu_SDA   var   PORTC.4      ' pin RC4
Fpu_ID    con   $C8          ' uM-FPU I2C device address
```



Brief Overview of the uM-FPU V3.1 Floating Point Coprocessor

For a full description of the uM-FPU V3.1 chip, please refer to the *uM-FPU V3.1 Datasheet*, *uM-FPU V3.1 Instruction Reference*. Application notes are also available on the Micromega website.

The uM-FPU V3.1 chip is a separate coprocessor with its own set of registers and instructions designed to provide microcontrollers with 32-bit floating point and long integer capabilities. The Microchip PIC communicates with the FPU using an SPI or I^2C interface. Instructions and data are sent to the FPU, and the FPU performs the calculations. The Microchip PIC is free to do other tasks while the FPU performs calculations. Results can be read back to the Microchip PIC or stored on the FPU for later use. The uM-FPU V3.1 chip has 128 registers, numbered 0 through 127, that can hold 32-bit floating point or long integer values. Register 0 is often used as a temporary register and is modified by some of the uM-FPU V3.1 instructions. Registers 1 through 127 are available for general use.

The `SELECTA` instruction is used to select any one of the 128 registers as register A. Register A can be regarded as an accumulator or working register. Arithmetic instructions use the value in register A as an operand and store the result of the operation in register A. If an instruction requires more than one operand, the additional operand is specified by the instruction. The following example selects register 2 as register A and adds register 5 to it:

```
SELECTA, 2          select register 2 as register A
FADD, 5             register[A] = register[A] + register[5]
```

Sending Instructions to the uM-FPU

Appendix A contains a table that gives a summary of each uM-FPU V3.1 instruction, with enough information to follow the examples in this document. For a detailed description of each instruction, refer to the *uM-FPU V3.1 Instruction Reference*.

To send instructions to the FPU using a SPI interface, the `SHIFTOUT` command is used as follows:

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FADD, 5]
```

To send instructions to the FPU using an I^2C interface, the `I2CWRITE` command is used as follows:

```
I2CWRITE Fpu_SDA, Fpu_SCL, Fpu_ID, 0, [FADD, 5]
```

The instructions and data to send to the FPU are located inside the square brackets. The beginning of the command is always the same, and depends on whether you are using an SPI or I^2C interface. It tells the Microchip PIC how to communicate with the FPU. The `SHIFTOUT` command is used for the examples in this document, but the `I2CWRITE` command would be substituted if an I^2C interface is used.

All instructions have an opcode that tells the FPU which operation to perform, The following example calculates the square root of register A:

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SQRT]
```

Some instructions require additional operands or data and are specified by the bytes following the opcode. The following example adds register 5 to register A.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FADD, 5]
```

Some instructions return data. This example reads the lower 8 bits of register A:

```
GOSUB Fpu_Wait
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [LREADBYTE]
GOSUB Fpu_ReadDelay
SHIFTIN FpuOut, FpuClk, MSBPRE, [dataByte]
```

The following example adds the value in register 5 to the value in register 2.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, 2, FADD, 5]
```

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

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

SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, Total, FADD, Count]
```

Tutorial Examples

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

Most of the data read from devices connected to the Microchip PIC 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 variable on the Microchip PIC called `rawTemp`. The value returned by the DS1620 is the temperature in units of 1/2 degrees Celsius. The following instructions load the `rawTemp` value to the FPU, convert it to floating point, then divide it by 2 to get degrees in Celsius.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST,
 [SELECTA, DegC, LOADWORD, rawTemp>>8, rawTemp, FSET0, FDIVI, 2]
```

Description:

`SELECTA, DegC`	select DegC as register A
`LOADWORD, rawTemp>>8, rawTemp`	load rawTemp to register 0 and convert to floating point
`FSET0`	DegC = register[0] (i.e. the floating point value of rawTemp)
`FDIVI, 2`	divide by the floating point value 2.0

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

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, F1_8, ATOF, "1.8", 0, FSET0]
```

Description:

`SELECTA, F1.8`	select F1_8 as register A
`ATOF, "1.8", 0`	load the string 1.8 (note: the string must be zero terminated), convert the string to floating point, and store in register 0
`FSET0`	F1_8 = register[0] (i.e. 1.8)

We calculate the degrees in Fahrenheit (F = C * 1.8 + 32) as follows:

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST,
 [SELECTA, DegF, FSET, DegC, FMUL, F1_8, FADDI, 32]
```

Description:

`SELECTA, DegF`	select DegF as register A
`FSET, DegC`	DegF = DegC
`FMUL, F1_8`	DegF = DegF * 1.8
`FADDI, 32`	DegF = DegF + 32.0

Note: this tutorial example is intended to show how to perform a familiar calculation, but the `FCNV` instruction could be used to perform unit conversions in one step. See the *uM-FPU V3.1 Instruction Reference* for a full list of conversions.

There are support routines provided for sending floating point and long integer strings to the debug port. `Print_Float` sends an unformatted floating point string and displays up to eight digits of precision. `Print_FloatFormat` sends a formatted floating point string. We'll use `Print_FloatFormat` to send the results to the debug port. The `format` variable is used to select the desired format, with the tens digit specifying the total number of characters to display, and the ones digit specifying 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_FloatFormat` routine the FPU register is selected and the `format` variable is set. The following example sends the temperature in degrees Celsius and Fahrenheit to the debug port.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, DegC]
format = 51
GOSUB Print_FloatFormat

SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, DegF]
format = 51
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 V3.1 Support Software

Support routines and opcode definitions for uM-FPU V3.1 are contained in two include files: *umfpuV3-spi.bas* (for SPI connections), and *umfpuV3-i2c.bas* (for I^2C connections) . To add uM-FPU V3.1 support to a PICBASIC PRO program, the main program should:

- define the oscillator speed
- define the debug port
- define the uM-FPU V3.1 pins
- include the SPI or I^2C support routines

Sample code for an SPI interface is as follows:

```
'-------------------- PIC oscillator speed -----------------------------------
define      OSC 20                    ' specify the speed of the oscillator
OSC_SPEED   con 20                    ' define PICBASIC constant

'-------------------- debug definitions --------------------------------------
define      DEBUG_REG    PORTC
define      DEBUG_BIT    6
defin       DEBUG_BAUD   19200
define      DEBUG_MODE   0

'-------------------- uM-FPU pin definitions ---------------------------------
FpuClk      var     PORTC.3           ' SPI SCLK (uM-FPU SCLK)
FpuIn       var     PORTC.4           ' SPI MISO (uM-FPU SOUT)
FpuOut      var     PORTC.5           ' SPI MOSI (uM-FPU SIN)

include     "umfpuV3-spi.bas"         ' include the uM-FPU V3 support routines
```

Sample code for an I^2C interface is as follows:

```
'-------------------- PIC oscillator speed -----------------------------------
define      OSC 20                    ' specify the speed of the oscillator
OSC_SPEED   con 20                    ' define PICBASIC constant

'-------------------- debug definitions --------------------------------------
define      DEBUG_REG    PORTC
define      DEBUG_BIT    6
defin       DEBUG_BAUD   19200
define      DEBUG_MODE   0


'-------------------- uM-FPU pin definitions ---------------------------------
Fpu_SCL     var    PORTC.3            ' I2C SCL pin (uM-FPU SCL)
Fpu_SDA     var    PORTC.4            ' I2C SDA pin (uM-FPU SDA)
Fpu_ID      con          $C8          ' uM-FPU I2C device address
define      I2C_HOLD     1            ' required for correct I2C timing

include     "umfpuV3-i2c.bas"         ' include the uM-FPU V3 support routines
```

The include files contain all of the support routines described below. Program template files (called *template-spi.bas* and *template-i2c.bas*) and various sample programs are provided for both SPI and I^2C.

Fpu_Reset

To ensure that the Microchip PIC and the FPU are synchronized, a reset call must be done at the start of every program. The `Fpu_Reset` routine resets the FPU, confirms communications, and returns the sync character ($5C) in the `fpu_status` variable if the reset is successful. A sample reset call is included in the *template-spi.bas* and *template-i2c.bas* files.

Fpu_Wait

The FPU must have completed all instructions in the instruction buffer, and be ready to return data, before sending an instruction to read data from the FPU. The `Fpu_Wait` routine checks the ready status of the FPU and waits until it is ready. The print routines check the ready status, so calling `Fpu_Wait` before calling a print routine isn't required, but if your program reads directly from the FPU using the `SHIFTIN` or `I2CREAD` commands, 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
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [LREADBYTE]
GOSUB Fpu_ReadDelay
SHIFTIN FpuOut, FpuClk, MSBPRE, [dataByte]
```

Description:

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

The uM-FPU V3.1 chip has a 256 byte instruction buffer. In most cases, data will be read back before 256 bytes have been sent to the FPU. If a long calculation is done that requires more than 256 bytes to be sent to the FPU, an `Fpu_Wait` call should be made at least every 256 bytes to ensure that the instruction buffer doesn't overflow.

Fpu_ReadStatus

The current status byte is read from the FPU and returned in the `fpu_status` variable.

Fpu_ReadDelay

After a read instruction is sent, and before the first data is read, a setup delay is required to ensure that the FPU is ready to send data. The `Fpu_ReadDelay` routine provides the required read setup delay. The delay is only required before the first byte read after a read instruction.

Print_Version

The FPU version string is sent to the debug port using the `DEBUG` command.

Print_Float

The value in register A is sent to the debug port as a floating point string using the `DEBUG` 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 sent to the debug port as a formatted floating point string using the `DEBUG` 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 sent to the debug port as a signed long integer string using the `DEBUG` 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 sent to the debug port as a formatted long integer string using the `DEBUG` 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

Print_FpuString

The contents of the FPU string buffer are sent to the debug port using the `DEBUG` command.

Loading Data Values to the FPU

Most of the data read from devices connected to the Microchip PIC will return some type of integer value. There are several ways to load integer values to the FPU and convert them to 32-bit floating point or long integer values.

8-bit Integer to Floating Point

The FSETI, FADDI, FSUBI, FSUBRI, FMULI, FDIVI, FDIVRI, FPOWI, and FCMPI instructions read the byte following the opcode as an 8-bit signed integer, convert the value to floating point, and then perform the operation. It's a convenient way to work with constants or data values that are signed 8-bit values. The following example stores the lower 8 bits of variable dataByte to the Result register on the FPU.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, Result, FSETI, dataByte]
```

The LOADBYTE instruction reads the byte following the opcode as an 8-bit signed integer, converts the value to floating point, and stores the result in register 0.

The LOADUBYTE instruction reads the byte following the opcode as an 8-bit unsigned integer, converts the value to floating point, and stores the result in register 0.

16-bit Integer to Floating Point

The LOADWORD instruction reads the two bytes following the opcode as a 16-bit signed integer (MSB first), converts the value to floating point, and stores the result in register 0. The following example adds the lower 16 bits of variable dataWord to the Result register on the FPU.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST,
 [SELECTA, Result, LOADWORD, dataWord>>8, dataWord, FADD0]
```

The LOADUWORD instruction reads the two bytes following the opcode as a 16-bit unsigned integer (MSB first), converts the value to floating point, and stores the result in register 0.

32-bit Floating Point to Floating point

The FWRITE, FWRITEA, FWRITEX, and FWRITE0 instructions interpret the four bytes following the opcode as a 32-bit floating point value and stores the value in the 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). The *uM-FPU V3 IDE* can be used to easily generate the 32-bit values. This example sets Angle = 20.0 (the floating point representation for 20.0 is hex 41A00000).

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FWRITE, Angle, $41, $A0, $00, $00]
```

ASCII string to Floating Point

The ATOF instruction is used to convert zero-terminated strings to floating point values. The instruction reads the bytes following the opcode (until a zero terminator is read), converts the string to floating point, and stores the result in register 0. The following example sets the register Angle to 1.5885.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, Angle, ATOF, "1.5885",0, FSET0]
```

8-bit Integer to Long Integer

The LSETI, LADDI, LSUBI, LMULI, LDIVI, LCMPI, LUDIVI, LUCMPI, and LTSTI instructions read the byte following the opcode as an 8-bit signed integer, convert the value to long integer, and then perform the operation. It's a convenient way to work with constants or data values that are signed 8-bit values. The following example adds the lower 8 bits of variable dataByte to the Total register on the FPU.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, Total, LADDI, dataByte]
```

The `LONGBYTE` instruction reads the byte following the opcode as an 8-bit signed integer, converts the value to long integer, and stores the result in register 0.

The `LONGUBYTE` instruction reads the byte following the opcode as an 8-bit unsigned integer, converts the value to long integer, and stores the result in register 0.

16-bit Integer to Long Integer

The `LONGWORD` instruction reads the two bytes following the opcode as a 16-bit signed integer (MSB first), converts the value to long integer, and stores the result in register 0. The following example adds the lower 16 bits of variable `dataWord` to the `Total` register on the FPU.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST,
 [SELECTA, Total, LOADWORD, dataWord>>8, dataWord, LADD0]
```

The `LONGUWORD` instruction reads the two bytes following the opcode as a 16-bit unsigned integer (MSB first), converts the value to long integer, and stores the result in register 0.

32-bit integer to Long Integer

The `LWRITE`, `LWRITEA`, `LWRITEX`, and `LWRITE0` instructions interpret the four bytes following the opcode as a 32-bit long integer value and stores the value in the specified register. This is used to load integer values greater than 16 bits. The *uM-FPU V3 IDE* can be used to easily generate the 32-bit values. For example, to set Total = 500000:

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [LWRITE, Total, $00, $07, $A1, $20]
```

ASCII string to Long Integer

The `ATOL` instruction is used to convert strings to long integer values. The instruction reads the bytes following the opcode (until a zero terminator is read), converts the string to long integer, and stores the result in register 0. The following example sets the register `Total` to 500000.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, Total, ATOL, "500000", 0, FSET0]
```

The fastest operations occur when the 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 128 registers available on the FPU, it’s often possible to pre-load all of the required constants. In non-critical sections of code, data and constants can be loaded as required.

Reading Data Values from the FPU

The uM-FPU V3.1 chip has a 256 byte instruction buffer which allows data transmission to continue while previous instructions are being executed. Before reading data, you must check to ensure that the previous commands have completed, and the FPU is ready to send data. The `Fpu_Wait` routine is used to wait until the FPU is ready, then a read command is sent, and the `SHIFTIN` or `I2CREAD` command is used to read data. The `Fpu_ReadDelay` routine must be called before the first byte is read after a read instruction.

8-bit Integer

The `LREADBYTE` instruction reads the lower 8 bits from register A. The following example stores the lower 8 bits of register A in variable `dataByte`.

```
GOSUB Fpu_wait
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [LREADBYTE]
GOSUB Fpu_ReadDelay
SHIFTIN FpuOut, FpuClk, MSBPRE, [dataByte]
```

16-bit Integer

The `LREADWORD` instruction reads the lower 16 bits from register A. The following example stores the lower 16 bits of register A in variable `dataWord.`

```
GOSUB Fpu_wait
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [LREADWORD]
GOSUB Fpu_ReadDelay
SHIFTIN FpuOut, FpuClk, MSBPRE, [tmp1, tmp2]
dataWord = tmp1<<8 + tmp2
```

Long Integer to ASCII string

The `LTOA` instruction can be used to convert long integer values to an ASCII string. The `Print_Long` and `Print_LongFormat` routines use this instruction to read the value from register A and send the long integer string to the debug port.

Floating Point

The `Fpu_ReadFloat` routine can be used to read 32-bit floating point values from the FPU. The following example reads the 32-bit floating point value from register A, and returns the value in the `tmp` variable.

```
GOSUB Fpu_wait
reg = -1
gosub Fpu_ReadFloat
```

Floating Point to ASCII string

The `FTOA` instruction can be used to convert floating point values to an ASCII string. The `Print_Float` and `Print_FloatFormat` routines use this instruction to read the value from register A and send the floating point string to the debug port.

Comparing and Testing Floating Point Values

Floating point values can be zero, positive, negative, infinite, or Not a Number (which occurs if an invalid operation is performed on a floating point value). The status byte is read using the `Fpu_ReadStatus` routine. It waits for the FPU to be ready before sending the `READSTATUS` instruction and reading the status byte. The current status is returned in the `fpu_status` variable. Bit definitions are provided for the status bits in the `fpu_status` variable as follows:

`fpu_status_Zero`	Zero status bit (0-not zero, 1-zero)
`fpu_status_Sign`	Sign status bit (0-positive, 1-negative)
`fpu_status_NaN`	Not a Number status bit (0-valid number, 1-NaN)
`fpu_status_Inf`	Infinity status bit (0-not infinite, 1-infinite)

The `FSTATUS` and `FSTATUSA` instructions are used to set the status byte to the floating point status of the selected register. The following example checks the floating point status of register A:

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FSTATUSA]
GOSUB Fpu_ReadStatus
IF (fpu_status_Sign = 1) THEN DEBUG "Result is negative"
IF (fpu_status_Zero = 1) THEN DEBUG "Result is zero"
```

The `FCMP`, `FCMP0`, and `FCMPI` instructions are used to compare two floating point values. The status bits are set for the result of register A minus the operand (the selected registers are not modified). For example, to compare register A to the value 10.0:

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FCMPI, 10]
GOSUB Fpu_ReadStatus
IF (fpu_status_Zero = 1) THEN
  DEBUG "Value1 = Value2"
ELSEIF (fpu_status_Sign = 1) THEN
  DEBUG "Value1 < Value2"
ELSE
  DEBUG "Value1 > Value2"
ENDIF
```

The `FCMP2` instruction compares two floating point registers. The status bits are set for the result of the first register minus the second register (the selected registers are not modified). For example, to compare registers `Value1` and `Value2`:

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FCMP2, Value1, Value2]
GOSUB Fpu_ReadStatus
```

Comparing and Testing Long Integer Values

A long integer value can be zero, positive, or negative. The status byte is read using the `Fpu_Status` routine. It waits for the FPU to be ready before sending the `READSTATUS` instruction and reading the status byte. The current status is returned in the `fpu_status` variable. Bit definitions are provided for the status bits in the `fpu_status` variable as follows:

`fpu_status_Zero`	Zero status bit (0-not zero, 1-zero)
`fpu_status_Sign`	Sign status bit (0-positive, 1-negative)

The `LSTATUS` and `LSTATUSA` instructions are used to set the status byte to the long integer status of the selected register. The following example checks the long integer status of register A:

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [LSTATUSA]
GOSUB Fpu_ReadStatus
IF (fpu_status_Sign = 1) THEN DEBUG "Result is negative"
IF (fpu_status_Zero = 1) THEN DEBUG "Result is zero"
```

The `LCMP`, `LCMP0`, and `LCMPI` instructions are used to do a signed comparison of two long integer values. The status bits are set for the result of register A minus the operand (the selected registers are not modified). For example, to compare register A to the value 10:

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [LCMPI, 10]
GOSUB Fpu_ReadStatus
IF (fpu_status_Zero = 1) THEN
  DEBUG "Value1 = Value2"
ELSEIF (fpu_status_Sign = 1) THEN
  DEBUG "Value1 < Value2"
ELSE
  DEBUG "Value1 > Value2"
ENDIF
```

The `LCMP2` instruction does a signed compare of two long integer registers. The status bits are set for the result of the first register minus the second register (the selected registers are not modified). For example, to compare registers `Value1` and `Value2`:

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [LCMP2, Value1, Value2]
GOSUB Fpu_ReadStatus
```

The `LUCMP`, `LUCMP0`, and `LUCMPI` instructions are used to do an unsigned comparison of two long integer values. The status bits are set for the result of register A minus the operand (the selected registers are not modified).

The `LUCMP2` instruction does an unsigned compare of two long integer registers. The status bits are set for the result of the first register minus the second register (the selected registers are not modified).

The `LTST`, `LTST0` and `LTSTI` instructions are used to do a bit-wise compare of two long integer values. The status bits are set for the logical `AND` of register A and the operand (the selected registers are not modified).

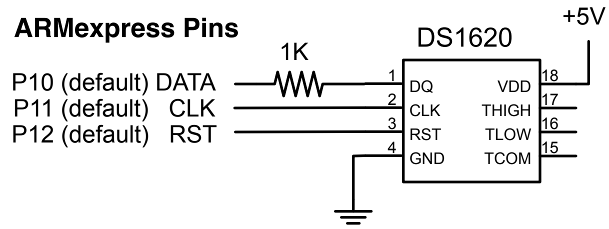
Further Information

The following documents are also available:

uM-FPU V3.1 Datasheet	provides hardware details and specifications
uM-FPU V3.1 Instruction Reference	provides detailed descriptions of each instruction
uM-FPU Application Notes	various application notes and examples

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

DS1620 Connections for Demo 1



Sample Code for Tutorial (demo1-spi.bas)

```
' This program demonstrates the use of the uM-FPU V3.1 floating point coprocessor
' with the PICBASIC PRO compiler using a API interface. It takes temperature
' readings from a DS1620 digital thermometer, converts them to floating point
' and displays them in degrees Celsius and degrees Fahrenheit.

'-------------------- PIC oscillator speed ----------------------------------
define       OSC 20                    ' specify the speed of the oscillator
OSC_SPEED    con 20                    ' define PICBASIC constant

'-------------------- debug definitions -------------------------------------
define       DEBUG_REG     PORTC
define       DEBUG_BIT     6
define       DEBUG_BAUD    19200
define       DEBUG_MODE    0

'-------------------- uM-FPU pin definitions --------------------------------
FpuClk       var    PORTC.3            ' SPI SCLK (uM-FPU SCLK)
FpuIn        var    PORTC.4            ' SPI MISO (uM-FPU SOUT)
FpuOut       var    PORTC.5            ' SPI MOSI (uM-FPU SIN)

include      "umfpuV3-spi.bas"           ' include uM-FPU V3.1 support routines

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

DS_RST       var    PORTA.0            ' DS1620 reset/enable
DS_CLK       var    PORTA.1            ' DS1620 clock
DS_DATA      var    PORTA.2            ' DS1620 data

LSBFIRS      con    0                  ' shiftout mode
LSBPRE       con    1                  ' shiftin mode

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

DegC         con    1                  ' degrees Celsius
DegF         con    2                  ' degrees Fahrenheit
F1_8         con    3                  ' constant 1.8

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

rawTemp      var    word               ' raw temperature reading
```

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

Reset:
  DEBUG 13, 10, 13, 10, "Demo 1: "

  GOSUB Fpu_Reset                       ' initialize uM-FPU
  IF fpu_status <> SYNC_CHAR THEN
    DEBUG "uM-FPU not detected."
    END
  ELSE
    GOSUB Print_Version                 ' display version string
  ENDIF
  DEBUG 13, 10, "---------------------"

  GOSUB Init_DS1620                     ' initialize DS1620

                                        ' load floating point constant
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, F1_8, ATOF, "1.8", 0, FSET0]

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

Main:
  GOSUB Read_DS1620                     ' get temperature reading from DS1620
  DEBUG 13, 10, 13, 10, "Raw Temp:  ", IHEX4 rawTemp

                                        ' send rawTemp to uM-FPU
                                        ' convert to floating point
                                        ' store in register
                                        ' divide by 2 to get degrees Celsius
  SHIFTOUT FpuOut, FpuClk, MSBFIRST,_
    [SELECTA, DegC, LOADWORD, rawTemp\16, FSET0, FDIVI, 2]

                                        ' degF = degC * 1.8 + 32
  SHIFTOUT FpuOut, FpuClk, MSBFIRST,_
    [SELECTA, DegF, FSET, DegC, FMUL, F1_8, FADDI, 32]

  DEBUG 13, 10, "Degrees C: "           ' display degrees Celsius
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, DegC]
  format = 51
  GOSUB Print_FloatFormat

  DEBUG 13, 10, "Degrees F: "           ' display degrees Fahrenheit
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, DegF]
  format = 51
  GOSUB Print_FloatFormat

  PAUSE 2000                            ' delay, then get the next reading
  GOTO Main
  END

'-------------------- Init_DS1620 ---------------------------------------------

Init_DS1620:
  ADCON1 = 6                            ' configure A0-A3 for digital I/O
```

```
  LOW DS_RST                          ' initialize pin states
  HIGH DS_CLK
  PAUSE 100

  HIGH DS_RST                         ' configure for CPU control
  SHIFTOUT DS_DATA, DS_CLK, LSBFIRST, [$0C, $02]
  LOW DS_RST
  PAUSE 100

  HIGH DS_RST                         ' start temperature conversions
  SHIFTOUT DS_DATA, DS_CLK, LSBFIRST, [$EE]
  LOW DS_RST
  PAUSE 1000                          ' wait for first conversion
  RETURN

'-------------------- Read_DS1620 --------------------------------------------

Read_DS1620:
  HIGH DS_RST                         ' read temperature value
  SHIFTOUT DS_DATA, DS_CLK, LSBFIRST, [$AA]
  SHIFTIN  DS_DATA, DS_CLK, LSBPRE, [rawTemp\9]
  LOW DS_RST                          ' extend the sign bit
  IF rawTemp.BIT8 = 1 THEN rawTemp.HIGHBYTE = $FF
  RETURN
```

Appendix A
uM-FPU V3.1 Instruction Summary

Instruction	Opcode	Arguments	Returns	Description
NOP	00			No Operation
SELECTA	01	nn		Select register A
SELECTX	02	nn		Select register X
CLR	03	nn		reg[nn] = 0
CLRA	04			reg[A] = 0
CLRX	05			reg[X] = 0, X = X + 1
CLR0	06			reg[nn] = 0
COPY	07	mm,nn		reg[nn] = reg[mm]
COPYA	08	nn		reg[nn] = reg[A]
COPYX	09	nn		reg[nn] = reg[X], X = X + 1
LOAD	0A	nn		reg[0] = reg[nn]
LOADA	0B			reg[0] = reg[A]
LOADX	0C			reg[0] = reg[X], X = X + 1
ALOADX	0D			reg[A] = reg[X], X = X + 1
XSAVE	0E	nn		reg[X] = reg[nn], X = X + 1
XSAVEA	0F			reg[X] = reg[A], X = X + 1
COPY0	10	nn		reg[nn] = reg[0]
COPYI	11	bb,nn		reg[nn] = long(unsigned byte bb)
SWAP	12	nn,mm		Swap reg[nn] and reg[mm]
SWAPA	13	nn		Swap reg[A] and reg[nn]
LEFT	14			Left parenthesis
RIGHT	15			Right parenthesis
FWRITE	16	nn,b1,b2,b3,b4		Write 32-bit floating point to reg[nn]
FWRITEA	17	b1,b2,b3,b4		Write 32-bit floating point to reg[A]
FWRITEX	18	b1,b2,b3,b4		Write 32-bit floating point to reg[X]
FWRITE0	19	b1,b2,b3,b4		Write 32-bit floating point to reg[0]
FREAD	1A	nn	b1,b2,b3,b4	Read 32-bit floating point from reg[nn]
FREADA	1B		b1,b2,b3,b4	Read 32-bit floating point from reg[A]
FREADX	1C		b1,b2,b3,b4	Read 32-bit floating point from reg[X]
FREAD0	1C		b1,b2,b3,b4	Read 32-bit floating point from reg[0]
ATOF	1E	aa…00		Convert ASCII to floating point
FTOA	1F	bb		Convert floating point to ASCII
FSET	20	nn		reg[A] = reg[nn]
FADD	21	nn		reg[A] = reg[A] + reg[nn]
FSUB	22	nn		reg[A] = reg[A] - reg[nn]
FSUBR	23	nn		reg[A] = reg[nn] - reg[A]
FMUL	24	nn		reg[A] = reg[A] * reg[nn]
FDIV	25	nn		reg[A] = reg[A] / reg[nn]
FDIVR	26	nn		reg[A] = reg[nn] / reg[A]
FPOW	27	nn		reg[A] = reg[A] ** reg[nn]
FCMP	28	nn		Compare reg[A], reg[nn], Set floating point status
FSET0	29			reg[A] = reg[0]
FADD0	2A			reg[A] = reg[A] + reg[0]

FSUB0	2B			reg[A] = reg[A] - reg[0]
FSUBR0	2C			reg[A] = reg[0] - reg[A]
FMUL0	2D			reg[A] = reg[A] * reg[0]
FDIV0	2E			reg[A] = reg[A] / reg[0]
FDIVR0	2F			reg[A] = reg[0] / reg[A]
FPOW0	30			reg[A] = reg[A] ** reg[0]
FCMP0	31			Compare reg[A], reg[0], Set floating point status
FSETI	32	bb		reg[A] = float(bb)
FADDI	33	bb		reg[A] = reg[A] - float(bb)
FSUBI	34	bb		reg[A] = reg[A] - float(bb)
FSUBRI	35	bb		reg[A] = float(bb) - reg[A]
FMULI	36	bb		reg[A] = reg[A] * float(bb)
FDIVI	37	bb		reg[A] = reg[A] / float(bb)
FDIVRI	38	bb		reg[A] = float(bb) / reg[A]
FPOWI	39	bb		reg[A] = reg[A] ** bb
FCMPI	3A	bb		Compare reg[A], float(bb), Set floating point status
FSTATUS	3B	nn		Set floating point status for reg[nn]
FSTATUSA	3C			Set floating point status for reg[A]
FCMP2	3D	nn,mm		Compare reg[nn], reg[mm] Set floating point status
FNEG	3E			reg[A] = -reg[A]
FABS	3F			reg[A] = I reg[A] I
FINV	40			reg[A] = 1 / reg[A]
SQRT	41			reg[A] = sqrt(reg[A])
ROOT	42	nn		reg[A] = root(reg[A], reg[nn])
LOG	43			reg[A] = log(reg[A])
LOG10	44			reg[A] = log10(reg[A])
EXP	45			reg[A] = exp(reg[A])
EXP10	46			reg[A] = exp10(reg[A])
SIN	47			reg[A] = sin(reg[A])
COS	48			reg[A] = cos(reg[A])
TAN	49			reg[A] = tan(reg[A])
ASIN	4A			reg[A] = asin(reg[A])
ACOS	4B			reg[A] = acos(reg[A])
ATAN	4C			reg[A] = atan(reg[A])
ATAN2	4D	nn		reg[A] = atan2(reg[A], reg[nn])
DEGREES	4E			reg[A] = degrees(reg[A])
RADIANS	4F			reg[A] = radians(reg[A])
FMOD	50	nn		reg[A] = reg[A] MOD reg[nn]
FLOOR	51			reg[A] = floor(reg[A])
CEIL	52			reg[A] = ceil(reg[A])
ROUND	53			reg[A] = round(reg[A])
FMIN	54	nn		reg[A] = min(reg[A], reg[nn])
FMAX	55	nn		reg[A] = max(reg[A], reg[nn])
FCNV	56	bb		reg[A] = conversion(bb, reg[A])
FMAC	57	nn,mm		reg[A] = reg[A] + (reg[nn] * reg[mm])
FMSC	58	nn,mm		reg[A] = reg[A] - (reg[nn] * reg[mm])

LOADBYTE	59	bb		reg[0] = float(signed bb)
LOADUBYTE	5A	bb		reg[0] = float(unsigned byte)
LOADWORD	5B	b1,b2		reg[0] = float(signed b1*256 + b2)
LOADUWORD	5C	b1,b2		reg[0] = float(unsigned b1*256 + b2)
LOADE	5D			reg[0] = 2.7182818
LOADPI	5E			reg[0] = 3.1415927
LOADCON	5F	bb		reg[0] = float constant(bb)
FLOAT	60			reg[A] = float(reg[A])
FIX	61			reg[A] = fix(reg[A])
FIXR	62			reg[A] = fix(round(reg[A]))
FRAC	63			reg[A] = fraction(reg[A])
FSPLIT	64			reg[A] = integer(reg[A]), reg[0] = fraction(reg[A])
SELECTMA	65	nn,b1,b2		Select matrix A
SELECTMB	66	nn,b1,b2		Select matrix B
SELECTMC	67	nn,b1,b2		Select matrix C
LOADMA	68	b1,b2		reg[0] = Matrix A[bb, bb]
LOADMB	69	b1,b2		reg[0] = Matrix B[bb, bb]
LOADMC	6A	b1,b2		reg[0] = Matrix C[bb, bb]
SAVEMA	6B	b1,b2		Matrix A[bb, bb] = reg[A]
SAVEMB	6C	b1,b2		Matrix B[bb, bb] = reg[A]
SAVEMC	6D	b1,b2		Matrix C[bb, bb] = reg[A]
MOP	6E	bb		Matrix/Vector operation
FFT	6F	bb		Fast Fourier Transform
WRBLK	70	tc t1…tn		Write multiple 32-bit values
RDBLK	71	tc	t1…tn	Read multiple 32-bit values
LOADIND	7A	nn		reg[0] = reg[reg[nn]]
SAVEIND	7B	nn		reg[reg[nn]] = reg[A]
INDA	7C	nn		Select register A using value in reg[nn]
INDX	7D	nn		Select register X using value in reg[nn]
FCALL	7E	bb		Call user-defined function in Flash
EECALL	7F	bb		Call user-defined function in EEPROM
RET	80			Return from user-defined function
BRA	81	bb		Unconditional branch
BRA	82	cc, bb		Conditional branch
JMP	83	b1, b2		Unconditional jump
JMP	84	cc, b1, b2		Conditional jump
TABLE	85	tc,t0…tn		Table lookup
FTABLE	86	cc,tc,t0…tn		Floating point reverse table lookup
LTABLE	87	cc,tc,t0…tn		Long integer reverse table lookup
POLY	88	tc,t0…tn		reg[A] = nth order polynomial
GOTO	89	nn		Computed GOTO
LWRITE	90	nn,b1,b2,b3,b4		Write 32-bit long integer to reg[nn]
LWRITEA	91	b1,b2,b3,b4		Write 32-bit long integer to reg[A]
LWRITEX	92	b1,b2,b3,b4		Write 32-bit long integer to reg[X], X = X + 1
LWRITE0	93	b1,b2,b3,b4		Write 32-bit long integer to reg[0]
LREAD	94	nn	b1,b2,b3,b4	Read 32-bit long integer from reg[nn]
LREADA	95		b1,b2,b3,b4	Read 32-bit long value from reg[A]

LREADX	96		b1,b2,b3,b4	Read 32-bit long integer from reg[X], X = X + 1
LREAD0	97		b1,b2,b3,b4	Read 32-bit long integer from reg[0]
LREADBYTE	98		bb	Read lower 8 bits of reg[A]
LREADWORD	99		b1,b2	Read lower 16 bits reg[A]
ATOL	9A	aa…00		Convert ASCII to long integer
LTOA	9B	bb		Convert long integer to ASCII
LSET	9C	nn		reg[A] = reg[nn]
LADD	9D	nn		reg[A] = reg[A] + reg[nn]
LSUB	9E	nn		reg[A] = reg[A] - reg[nn]
LMUL	9F	nn		reg[A] = reg[A] * reg[nn]
LDIV	A0	nn		reg[A] = reg[A] / reg[nn] reg[0] = remainder
LCMP	A1	nn		Signed compare reg[A] and reg[nn], Set long integer status
LUDIV	A2	nn		reg[A] = reg[A] / reg[nn] reg[0] = remainder
LUCMP	A3	nn		Unsigned compare reg[A] and reg[nn], Set long integer status
LTST	A4	nn		Test reg[A] AND reg[nn], Set long integer status
LSET0	A5			reg[A] = reg[0]
LADD0	A6			reg[A] = reg[A] + reg[0]
LSUB0	A7			reg[A] = reg[A] - reg[0]
LMUL0	A8			reg[A] = reg[A] * reg[0]
LDIV0	A9			reg[A] = reg[A] / reg[0] reg[0] = remainder
LCMP0	AA			Signed compare reg[A] and reg[0], set long integer status
LUDIV0	AB			reg[A] = reg[A] / reg[0] reg[0] = remainder
LUCMP0	AC			Unsigned compare reg[A] and reg[0], Set long integer status
LTST0	AD			Test reg[A] AND reg[0], Set long integer status
LSETI	AE	bb		reg[A] = long(bb)
LADDI	AF	bb		reg[A] = reg[A] + long(bb)
LSUBI	B0	bb		reg[A] = reg[A] - long(bb)
LMULI	B1	bb		reg[A] = reg[A] * long(bb)
LDIVI	B2	bb		reg[A] = reg[A] / long(bb) reg[0] = remainder
LCMPI	B3	bb		Signed compare reg[A] - long(bb), Set long integer status
LUDIVI	B4	bb		reg[A] = reg[A] / unsigned long(bb) reg[0] = remainder
LUCMPI	B5	bb		Unsigned compare reg[A] and long(bb), Set long integer status
LTSTI	B6	bb		Test reg[A] AND long(bb), Set long integer status
LSTATUS	B7	nn		Set long integer status for reg[nn]

LSTATUSA	B8			Set long integer status for reg[A]
LCMP2	B9	nn,mm		Signed long compare reg[nn], reg[mm] Set long integer status
LUCMP2	BA	nn,mm		Unsigned long compare reg[nn], reg[mm] Set long integer status
LNEG	BB			reg[A] = -reg[A]
LABS	BC			reg[A] = I reg[A] I
LINC	BD	nn		reg[nn] = reg[nn] + 1, set status
LDEC	BE	nn		reg[nn] = reg[nn] - 1, set status
LNOT	BF			reg[A] = NOT reg[A]
LAND	C0	nn		reg[A] = reg[A] AND reg[nn]
LOR	C1	nn		reg[A] = reg[A] OR reg[nn]
LXOR	C2	nn		reg[A] = reg[A] XOR reg[nn]
LSHIFT	C3	nn		reg[A] = reg[A] shift reg[nn]
LMIN	C4	nn		reg[A] = min(reg[A], reg[nn])
LMAX	C5	nn		reg[A] = max(reg[A], reg[nn])
LONGBYTE	C6	bb		reg[0] = long(signed byte bb)
LONGUBYTE	C7	bb		reg[0] = long(unsigned byte bb)
LONGWORD	C8	b1,b2		reg[0] = long(signed b1*256 + b2)
LONGUWORD	C9	b1,b2		reg[0] = long(unsigned b1*256 + b2)
SETSTATUS	CD	ss		Set status byte
SEROUT	CE	bb bb bd bb aa…00		Serial output
SERIN	CF	bb		Serial input
SETOUT	D0	bb		Set OUT1 and OUT2 output pins
ADCMODE	D1	bb		Set A/D trigger mode
ADCTRIG	D2			A/D manual trigger
ADCSCALE	D3	ch		ADCscale[ch] = B
ADCLONG	D4	ch		reg[0] = ADCvalue[ch]
ADCLOAD	D5	ch		reg[0] = float(ADCvalue[ch]) * ADCscale[ch]
ADCWAIT	D6			wait for next A/D sample
TIMESET	D7			time = reg[0]
TIMELONG	D8			reg[0] = time (long integer)
TICKLONG	D9			reg[0] = ticks (long integer)
EESAVE	DA	mm,nn		EEPROM[nn] = reg[mm]
EESAVEA	DB	nn		EEPROM[nn] = reg[A]
EELOAD	DC	mm,nn		reg[mm] = EEPROM[nn]
EELOADA	DD	nn		reg[A] = EEPROM[nn]
EEWRITE	DE	nn,bc,b1…bn		Store bytes in EEPROM
EXTSET	E0			external input count = reg[0]
EXTLONG	E1			reg[0] = external input counter
EXTWAIT	E2			wait for next external input
STRSET	E3	aa…00		Copy string to string buffer
STRSEL	E4	bb, bb		Set selection point
STRINS	E5	aa…00		Insert string at selection point
STRCMP	E6	aa…00		Compare string with string buffer
STRFIND	E7	aa…00		Find string and set selection point

STRFCHR	E8	aa…00		Set field separators
STRFIELD	E9	bb		Find field and set selection point
STRTOF	EA			Convert selected string to floating point
STRTOL	EB			Convert selected string to long integer
READSEL	EC		aa…00	Read selected string
STRBYTE	ED	bb		Insert byte at selection point
STRINC	EE			Increment string selection point
STRDEC	EF			Decrement string selection point
SYNC	F0		5C	Get synchronization byte
READSTATUS	F1		ss	Read status byte
READSTR	F2		aa…00	Read string from string buffer
VERSION	F3			Copy version string to string buffer
IEEEMODE	F4			Set IEEE mode (default)
PICMODE	F5			Set PIC mode
CHECKSUM	F6			Calculate checksum for uM-FPU code
BREAK	F7			Debug breakpoint
TRACEOFF	F8			Turn debug trace off
TRACEON	F9			Turn debug trace on
TRACESTR	FA	aa…00		Send string to debug trace buffer
TRACEREG	FB	nn		Send register value to trace buffer
READVAR	FC	nn		Read internal register value
RESET	FF			Reset (9 consecutive FF bytes cause a reset, otherwise it is a NOP)

Notes:

Opcode	Instruction opcode in hexadecimal
Arguments	Additional data required by instruction
Returns	Data returned by instruction
nn	register number (0-127)
mm	register number (0-127)
fn	function number (0-63)
bb	8-bit value
b1,b2	16-bit value (b1 is MSB)
b1,b2,b3,b4	32-bit value (b1 is MSB)
b1…bn	string of 8-bit bytes
ss	Status byte
bd	baud rate and debug mode
cc	Condition code
ee	EEPROM address slot (0-255)
ch	A/D channel number
bc	Byte count
tc	32-bit value count
t1…tn	String of 32-bit values
aa…00	Zero terminated ASCII string

In the FPUdefs.bas file, LEFT, RIGHT, READ, SIN, COS, GOTO, SEROUT, SERIN have been renamed to include an F_ prefix (e.g. F_SIN, F_COS, etc.) to avoid conflicts with reserved symbol names.

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 V3.1 is defined by the 32-bit IEEE 754 standard. The number of significant digits for a 32-bit floating point number is slightly more than 7 digits, and the range of values that can be handled is approximately $\pm 10^{38.53}$.

32-bit IEEE 754 Floating Point Representation

IEEE 754 floating point numbers have three components: a sign, exponent, the mantissa. The sign indicates whether the number is positive or negative. The exponent has an implied base of two and a bias value. The mantissa represents the fractional part of the number.

The 32-bit IEEE 754 representation is as follows:

Bit 31	30 – 23	22 – 0
S	Exponent	Mantissa

Sign Bit (bit 31)

The sign bit is 0 for a positive number and 1 for a negative number.

Exponent (bits 30-23)

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 (bits 30-23)

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 32-bit IEEE 754 floating point values displayed as four 8-bit hexadecimal constants are as follows:

```
$00, $00, $00, $00     ' 0.0
$3D, $CC, $CC, $CD     ' 0.1
$3F, $00, $00, $00     ' 0.5
$3F, $40, $00, $00     ' 0.75
$3F, $7F, $F9, $72     ' 0.9999
$3F, $80, $00, $00     ' 1.0
$40, $00, $00, $00     ' 2.0
$40, $2D, $F8, $54     ' 2.7182818 (e)
$40, $49, $0F, $DB     ' 3.1415927 (pi)
$41, $20, $00, $00     ' 10.0
$42, $C8, $00, $00     ' 100.0
$44, $7A, $00, $00     ' 1000.0
$44, $9A, $52, $2B     ' 1234.5678
$49, $74, $24, $00     ' 1000000.0
$80, $00, $00, $00     ' -0.0
$BF, $80, $00, $00     ' -1.0
$C1, $20, $00, $00     ' -10.0
$C2, $C8, $00, $00     ' -100.0
$7F, $C0, $00, $00     ' NaN (Not-a-Number)
$7F, $80, $00, $00     ' +inf
$FF, $80, $00, $00     ' -inf
```