



# Using the uM-FPU V2 Integrated Development Environment (IDE)

**Micromega Corporation**

## Introduction

The uM-FPU Integrated Development Environment (IDE) software provides an easy-to-use tool for developing applications using the uM-FPU floating point coprocessor. The IDE runs on Windows 98, NT, ME, 2000 and XP, and provides support for:

- Generating uM-FPU code for math expressions
- Debugging uM-FPU code
- Storing user-defined functions on the uM-FPU chip

The code generator is a type of application builder, or wizard, that takes standard math expressions and generates the required uM-FPU code for a selected target. A number of targets are supported, including: Basic Stamp®, Javelin Stamp™, SX/B Compiler, PICAXE and PICmicro® assembler. Additional targets will be added – check the Micromega website for up-to-date information. The generated code can be easily copy and pasted into the user's microcontroller program in the target development environment. The examples shown in this document use the BASIC Stamp with an SPI interface. If you are working with a different microcontroller or compiler, the procedures are the same, but the output code and target development environment will be different.

The IDE uses the built-in debugger on the uM-FPU to provide the user with valuable debugging information such as uM-FPU instruction traces, and register values, and allows the user to set breakpoints and step through the execution of uM-FPU instructions.

The IDE supports storing user-defined functions and setting parameters on the uM-FPU chip. This unique capability can reduce the memory usage on the microcontroller, simplify the interface, and increase the speed of operations.

The IDE takes a source file containing symbol definitions and math expressions. When the source file is compiled, output code is generated for the target, symbol definitions are stored in the debugger, and any functions that are defined are stored in a function list – ready to be programmed into Flash memory on the uM-FPU. The source file is a text file with a default filename extension of *fpu*.

Debugging and function programming requires a serial connection between the PC running the IDE and the uM-FPU chip. Code generation doesn't require a connection. There are various ways of providing a serial connection, see *Application Note 9 – Adding a Serial Connection to the uM-FPU V2* for more information.

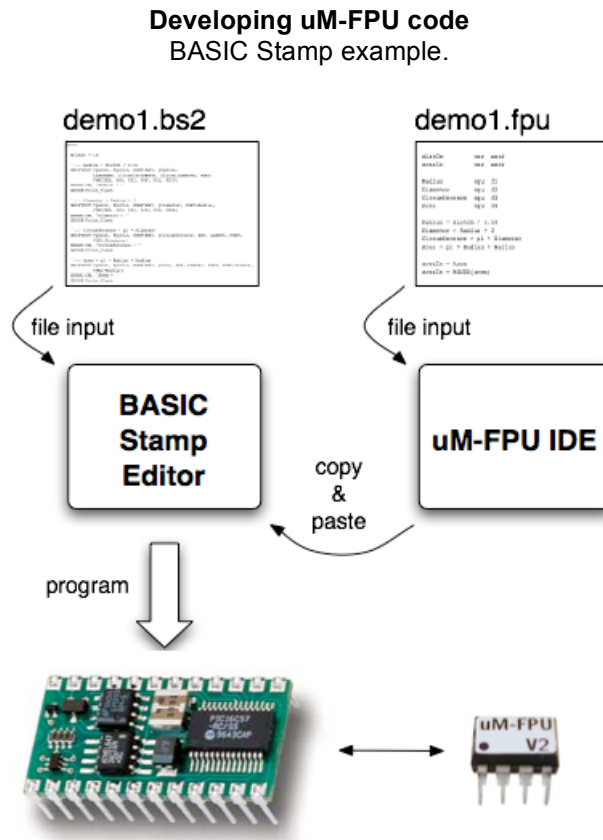
This document contains the following section:

- Overview of Using the IDE to Generate uM-FPU Code
- Tutorial 1 – Generating uM-FPU Code
- Tutorial 2 - Debugging uM-FPU Code
- Tutorial 3 - Storing User-Defined Functions
- Reference Guide for Generating uM-FPU Code
- Reference Guide for Debugging
- Reference Guide for Storing User-Defined Functions
- Reference Guide for Setting uM-FPU Parameters

The tutorials demonstrate how to use the uM-FPU IDE by going through some simple examples. The reference guides provide detailed information on the features of the IDE.

## Overview of Using the IDE to Generate uM-FPU Code

The figure below shows the process of developing uM-FPU code using the IDE. The left side of the diagram shows the normal development process. The user enters a BASIC Stamp program (called *demo1.bs2* in this example), compiles it, and programs the BASIC Stamp using the Parallax BASIC Stamp Editor. The right side of the diagram shows the additional steps for generating uM-FPU code. The user enters the uM-FPU source file (called *demo1.fpu* in this example), compiles it with the IDE, then copies the generated code from the IDE, and pastes it into the BASIC Stamp program. The program is then compiled and run as usual with the BASIC Stamp Editor.



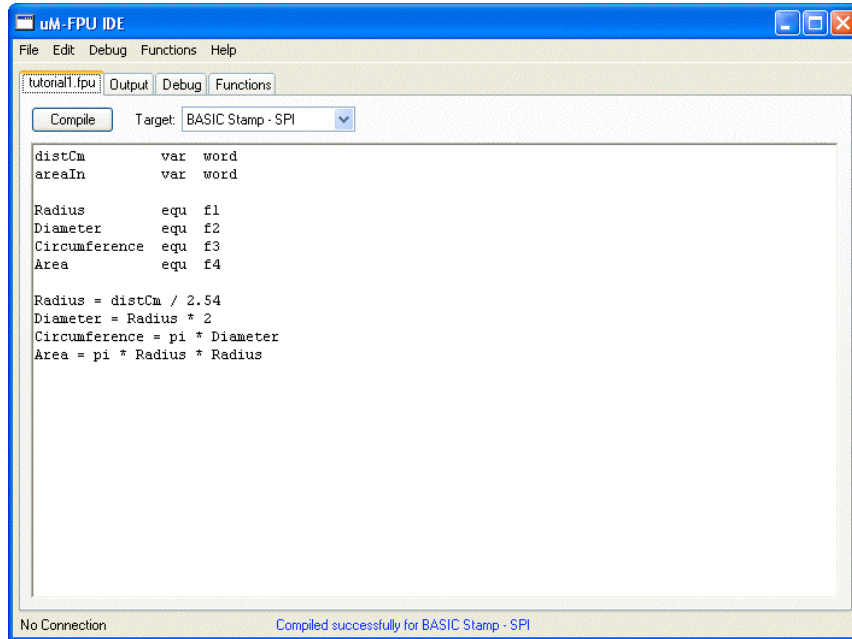
The IDE allows you to specify definitions and math expressions using very traditional looking code. It generates the uM-FPU instructions for you and outputs them in the compiled format of the selected target.

The IDE uses an interface that consists of a main window with tabs to select one of the following windows:

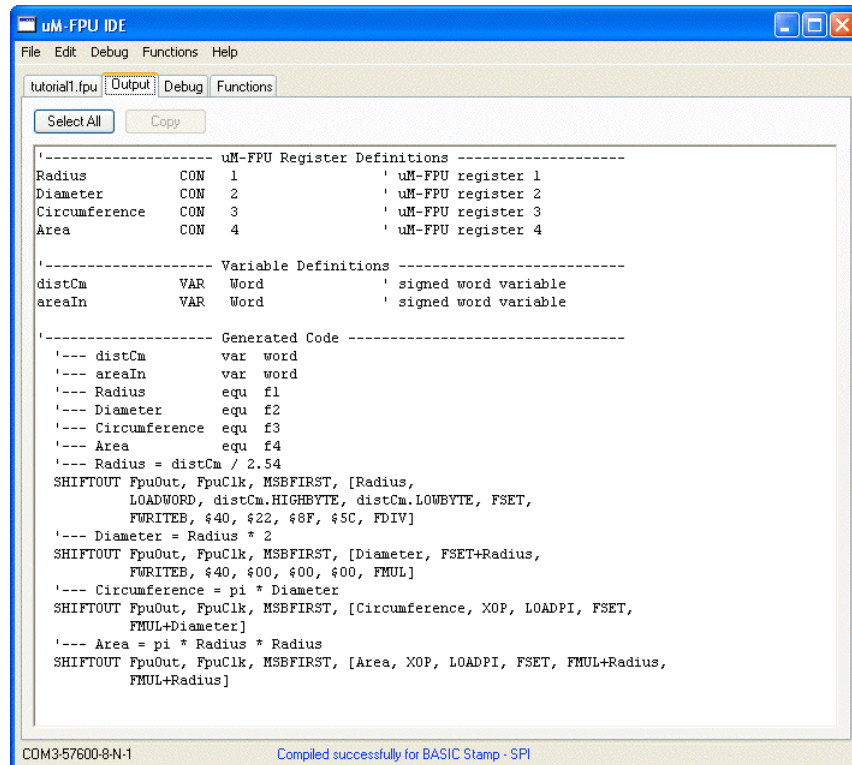
- Source File
- Output
- Debug
- Functions

Clicking the tab will display the associated window. The source file is associated with the leftmost tab, and the name of the source file is displayed on the tab. If the source file has not been previously saved, the name of the tab will be *untitled*. An example of the four window types is shown below.

**Source File Window**  
Used to display and edit the source file.

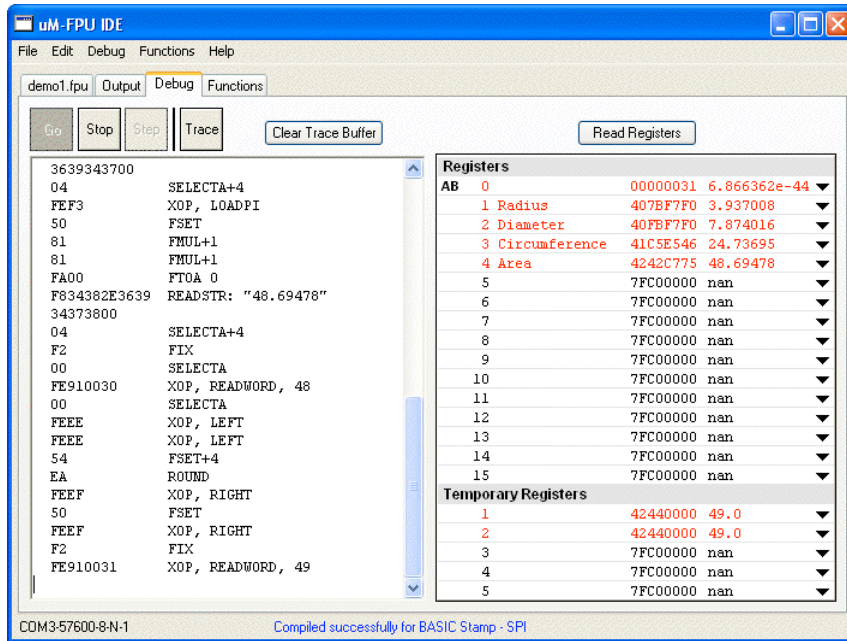


**Output Window**  
Used to display the generated output code.



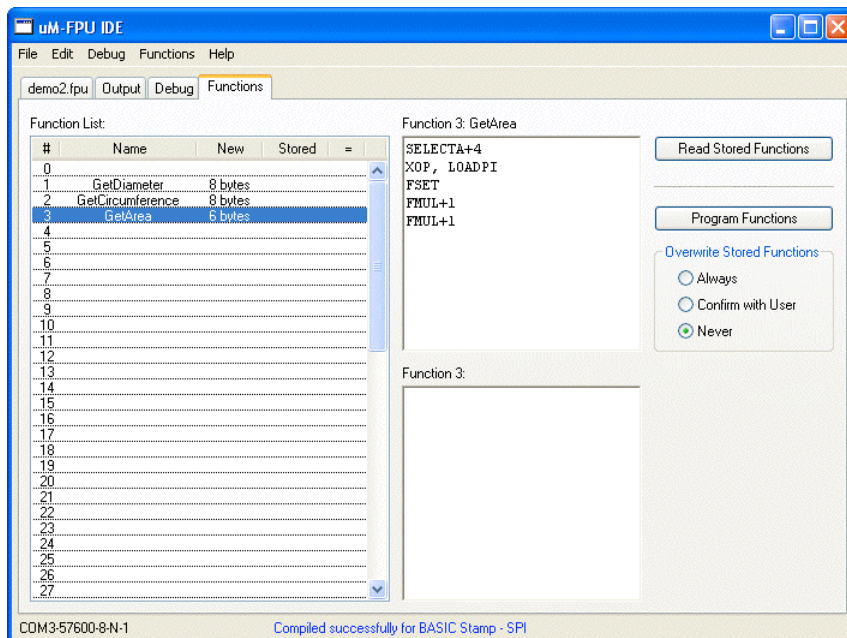
### Debug Window

Used for debugging uM-FPU code.



### Functions Window

Used to store user-defined functions.



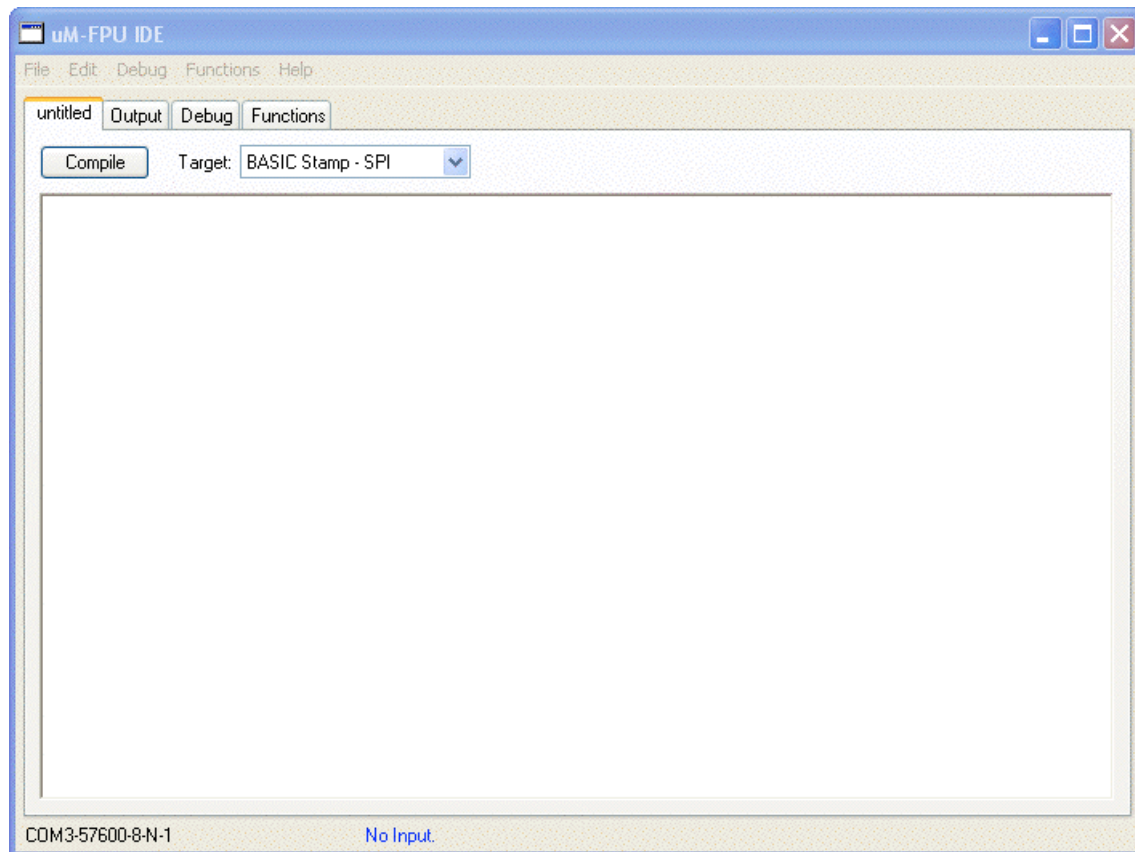
## Tutorial 1 – Generating uM-FPU Code

This tutorial takes you through the process of generating code for a simple example. Various IDE features are introduced as we go through the tutorial. For a more complete description of specific features, see the section entitled *Reference Guide for Generating uM-FPU Code*.

### Starting the uM-FPU IDE

Start the uM-FPU IDE program. The program will open with an empty source file called *untitled* as shown in the figure below. The **Target** pop-up menu is used to select the desired target for the output code. We will use **BASIC Stamp – SPI** in this tutorial. The connection status is shown at the lower left of the window. It shows the port, baud rate and format of the serial connection. A connection is only required for debugging and storing user-defined functions. You can use the **Select Port...** item in the **Debug** or **Functions** menu to select the desired port or to select **No Connection**. The program status is displayed at the bottom of the window, and should now be displaying *No Input*.

**Source File Window**  
As displayed at start of program.

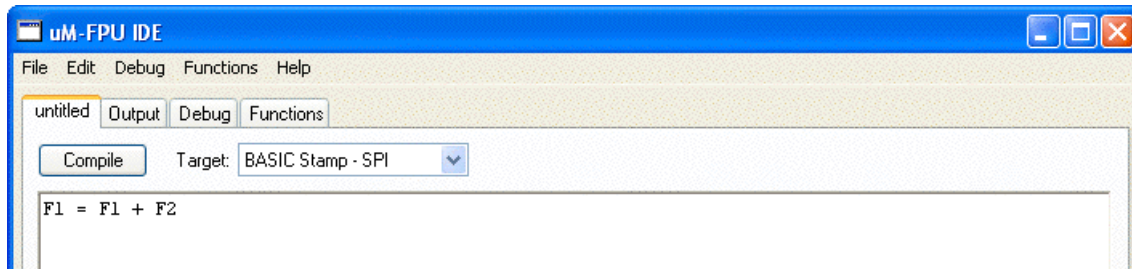


### A Quick Introduction to Generating Code

The IDE uses predefined symbols for the 16 registers in the uM-FPU. The pre-defined symbols **F0**, **F1**, **F2**, ... **F15** are used to specify registers 0 through 15 and define the type as floating point. To add the floating point value in register 2, to the floating point value in register 1, we would use the following expression:

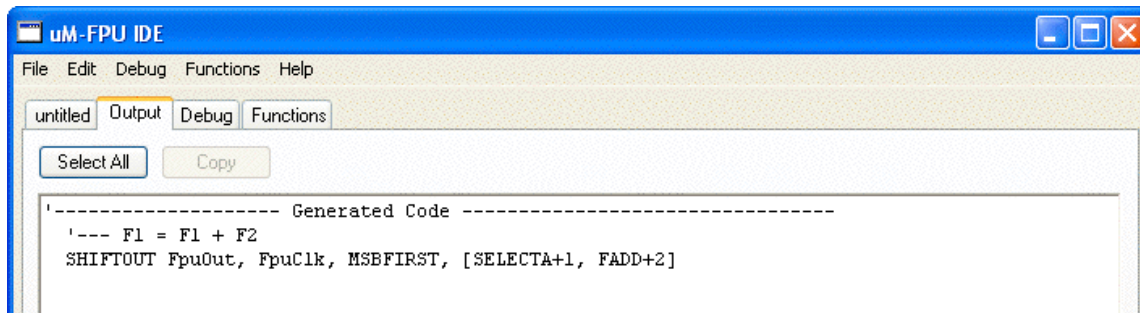
$$F1 = F1 + F2$$

Type this expression into the source file as shown below.



Notice that the status line at the bottom of the window now reads **Input modified since last compile**. This lets you know that you must compile to get up-to-date output code. Click on the **Compile** button. The status should change to **Compiled successfully for BASIC Stamp – SPI**. If an error is detected, an error message will be displayed in red. If you get an error message, check that your input matches the figure above, then click the **Compile** button again.

Now click the **Output** tab and the following code should be displayed:



The expression  $F1 = F1 + F2$  has been translated into BASIC Stamp code to select uM-FPU register 1 as the A register, then add the value of register 2 to the A register. You've successfully compiled your first expression.

### Defining Symbols

The IDE allows you to define symbols for uM-FPU registers, microcontroller variables and constants. Defining symbols with meaningful names makes it easier to read and understand expressions.

Registers are defined using the EQU operator and one of the predefined register symbols. For example, the following statements define the symbol TOTAL as a floating point value in register 1, and COUNT as a byte variable on the microcontroller.

```
TOTAL EQU F1
COUNT VAR BYTE
```

The following statement would generate code to read the value of COUNT from the microcontroller, convert it to floating point and add it to the TOTAL register.

```
TOTAL = TOTAL + COUNT
```

## A Sample Project

We have a distance measuring device that returns a number of pulses proportional to distance. It can measure distance from 0 to 30 inches and returns 1000 pulses per inch. We intend to use this device to measure the radius of a circle, calculate the diameter, circumference and area of the circle, and display the results in units of inches to three decimal places.

## Calculating Radius

The number of pulses returned by the distance measuring device will range from 0 to 30000 (30 inches x 1000 pulses per inch), so we will need to use a word variable to store the value on the microcontroller. Since results will be displayed in inches, once the distance is loaded to the uM-FPU as a floating point number we'll divide the value by 1000.

Enter the following code and Click the  button.

```
distance  VAR  word
Radius    EQU  F1

Radius = distance / 1000
```

The output is shown below:

```
'----- uM-FPU Register Definitions -----
Radius          CON  1          ' uM-FPU register 1
'----- Variable Definitions -----
distance        VAR  Word      ' signed word variable
'----- Generated Code -----
'--- distance  VAR  word
'--- Radius    EQU  F1
'--- Radius = distance / 1000
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [Radius,
        LOADWORD, distance.HIGHBYTE, distance.LOWBYTE, FSET,
        FWRITEB, $44, $7A, $00, $00, FDIV]
```

The Radius register is selected as the A register; the LOADWORD instruction loads the 16-bit value of distance (from the microcontroller) and converts it to floating point; the FSET instruction sets Radius equal to the distance value; the FWRITEB instruction loads the floating point value of 1000, and the FDIV instruction divides Radius by 1000.

## Copy the Code to your BASIC Stamp Program

Open BASIC Stamp Editor, load the template file *umfpu-spi.bs2* and save a copy called *tutorial1.bs2*.

Copy the register definitions and variable definitions from the uM-FPU IDE and paste them into the *tutorial1.bs2* file in the BASIC Stamp Editor. The definitions should be placed after the main definitions comment line.

Since we don't actually have the sensor described, we will enter a test value at the start of the program. Add the following line immediately after the label called *Main*.

```
distance = 2575
```

Copy the generated code from the uM-FPU IDE and paste it into *tutorial1.bs2*.

To print the result, add the following lines immediately after the code you copied.

```
DEBUG CR, "Radius = "
GOSUB Print_Float
```

Your program should look like the following (not including the uM-FPU support code from the template):

```
'=====
'===== main definitions =====
'=====

'----- uM-FPU Register Definitions -----
Radius          CON  1          ' uM-FPU register 1

'----- Variable Definitions -----
distance        VAR  Word      ' signed word variable

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

Reset:
GOSUB Fpu_Reset          ' reset the FPU hardware
IF status <> SyncChar THEN
  DEBUG "uM-FPU not detected."
END
ELSE
  GOSUB Print_Version    ' display the uM-FPU version number
  DEBUG CR
ENDIF

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

Main:
distance = 2575

'--- Radius = distance / 1000
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [Radius,
LOADWORD, distance.HIGHBYTE, distance.LOWBYTE, FSET,
FWRITEB, $44, $7A, $00, $00, FDIV]

DEBUG CR, "Radius = "
GOSUB Print_Float
END
```

## Running the Program

Run the BASIC Stamp program. The following output should be displayed in the terminal window.



```
uM-FPU V2.0
Radius = 2.5749997
```

Notice that 2575 divided by 1000 is displayed as 2.5749997. The `Print_Float` routine displays the result with no formatting. Since we want to display the result with 3 decimal places, we will use the `Print_FloatFormat` routine. Replace:

```
GOSUB Print_Float
with
format = 63
GOSUB Print_FloatFormat
```



A format of 63 specifies a width of 6 characters and 3 decimal places. Run the program again. The displayed result should now be 2.575.

### Calculating Diameter, Circumference and Area

Now that we have the initial program, let's add the calculations for diameter, circumference and area. Add the following register definitions:

```
Diameter      equ  F2
Circumference equ  F3
Area          equ  F4
```

The area of a circle is twice the radius, so we add the following line to calculate diameter:

```
Diameter = Radius * 2
```

The circumference of a circle is equal to the value  $\pi$  (pi) times the diameter. The IDE has a pre-defined symbol for  $\pi$ , called PI, so you can simply enter the following line to calculate circumference.

```
Circumference = PI * Diameter
```

The area of a circle is equal to  $\pi$  times radius<sup>2</sup>. There is a POWER function that you could use to calculate radius to the power of 2, but for squared values, it is easier and more efficient to simply multiply the value by itself. Enter the following line to calculate the area:

```
Area = PI * Radius * Radius
```

Finally, we'll read the Area value back to the microcontroller as a 16-bit integer and print the result. To do this we add the following definition for the microcontroller variable:

```
areaIn      VAR  Word
```

Adding the following line will convert the Area value to long integer and send the lower 16-bits back to microcontroller.

```
areaIn = Area
```

With these new additions, the *tutorial.fpu* source file should now read as follows:

```
distance      VAR  Word
areaIn        VAR  Word

Radius        equ  F1
Diameter      equ  F2
Circumference equ  F3
Area          equ  F4

Radius = distance / 1000
Diameter = Radius * 2
Circumference = PI * Diameter
Area = PI * Radius * Radius

areaIn = Area
```

### Copy the Code to the BASIC Stamp Program

Compile the new code, copy the generated code from the uM-FPU IDE to the BASIC Stamp program, and add additional DEBUG statements to print the new results. Your BASIC Stamp program should now look like the following (not including the uM-FPU support code from the template):

```

=====
===== main definitions =====
=====

'----- uM-FPU Register Definitions -----
Radius          CON    1          ' uM-FPU register 1
Diameter        CON    2          ' uM-FPU register 2
Circumference   CON    3          ' uM-FPU register 3
Area            CON    4          ' uM-FPU register 4

'----- Variable Definitions -----
distance        VAR    Word       ' signed word variable
areaIn         VAR    Word       ' signed word variable

=====
----- initialization -----
=====

Reset:
  GOSUB Fpu_Reset                ' reset the FPU hardware
  IF status <> SyncChar THEN
    DEBUG "uM-FPU not detected."
  END
  ELSE
    GOSUB Print_Version          ' display the uM-FPU version number
  DEBUG CR
  ENDIF

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

Main:
  distance = 2575

  '--- Radius = distance / 1000
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [Radius,
    LOADWORD, distance.HIGHBYTE, distance.LOWBYTE, FSET,
    FWRITEB, $44, $7A, $00, $00, FDIV]
  DEBUG CR, "Radius:          "
  format = 63
  GOSUB Print_FloatFormat

  '--- Diameter = Radius * 2
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [Diameter, FSET+Radius,
    FWRITEB, $40, $00, $00, $00, FMUL]
  DEBUG CR, "Diameter:       "
  format = 63
  GOSUB Print_FloatFormat

  '--- Circumference = PI * Diameter
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [Circumference, XOP, LOADPI, FSET,
    FMUL+Diameter]
  DEBUG CR, "Circumference:  "
  format = 63
  GOSUB Print_FloatFormat

  '--- Area = PI * Radius * Radius
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [Area, XOP, LOADPI, FSET, FMUL+Radius,
    FMUL+Radius]
  DEBUG CR, "Area:           "
  format = 63
  GOSUB Print_FloatFormat

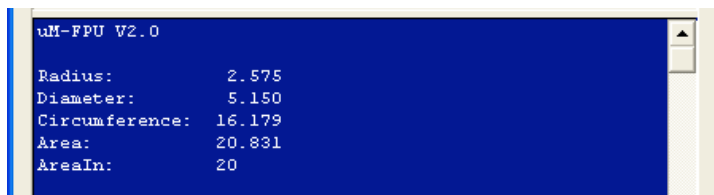
```

```
'--- areaIn = Area
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [Area]
GOSUB Fpu_Wait
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FIX, SELECTA]
GOSUB Fpu_Wait
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [XOP, READWORD]
SHIFTIN FpuIn, FpuClk, MSBPRES, [areaIn.HIGHBYTE, areaIn.LOWBYTE]
DEBUG CR, "AreaIn:      ", DEC AreaIn

END
```

## Running the Program

Run the BASIC Stamp program. The following output should be displayed in the terminal window:



```
uM-FPU V2.0
Radius:      2.575
Diameter:    5.150
Circumference: 16.179
Area:        20.831
AreaIn:      20
```

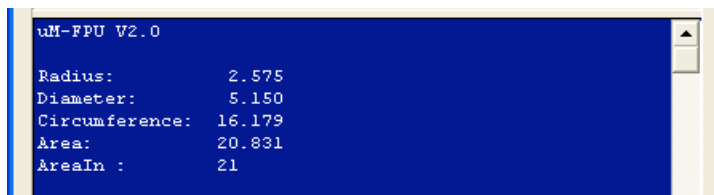
Notice that Area is displayed as 20.831, but areaIn is displayed as 20. This is because when a floating point number is converted to a long integer it is truncated, not rounded. If you prefer the value to be rounded, then use the ROUND function before converting the number. In the uM-FPU IDE, replace:

```
areaIn = Area
```

with:

```
areaIn = ROUND(area)
```

Compile the uM-FPU code, copy and paste the new code to the BASIC Stamp program, run the program again. The following output should now be displayed in the terminal window:



```
uM-FPU V2.0
Radius:      2.575
Diameter:    5.150
Circumference: 16.179
Area:        20.831
AreaIn :     21
```

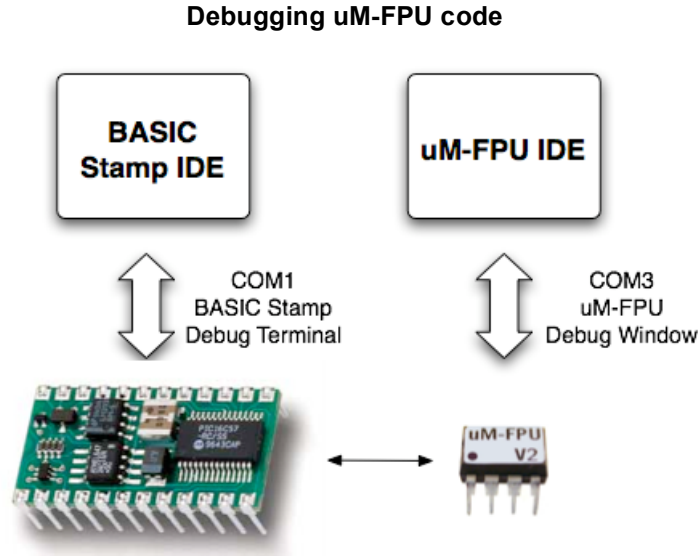
## Save the source file

Select the Save command from the File menu, enter the name *tutorial1.fpu* in the save dialog, and click the Save button.

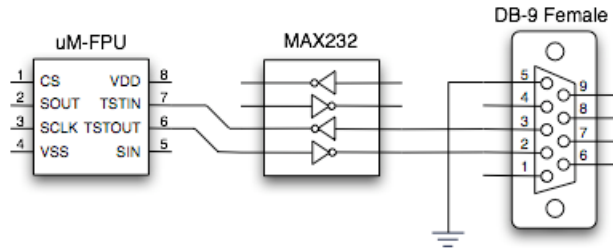
This completes the tutorial on generating uM-FPU code. Using the information gained from this tutorial, and the more detailed information from the reference section, you should now be able to use the IDE to create your own programs.

## Tutorial 2 - Debugging uM-FPU Code

The figure below shows the process of debugging uM-FPU code using the IDE. The left side of the diagram shows the normal BASIC Stamp debug connection using a serial port (COM1 in this example) and the terminal window in the BASIC Stamp Editor. The right side of the diagram shows the additional connection (COM3 in this example) used to connect the uM-FPU IDE to the built-in debugger on the uM-FPU chip.

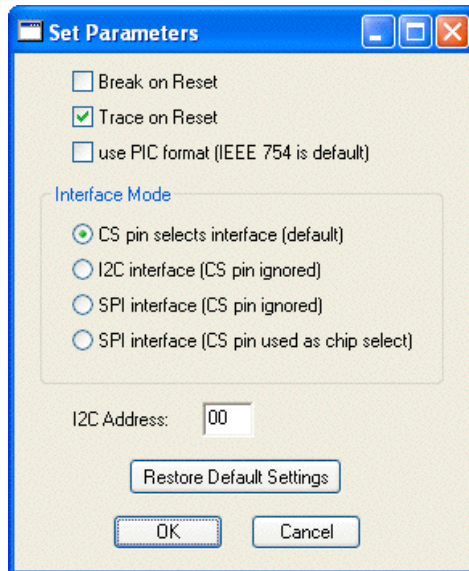


To use the debugger, the uM-FPU IDE requires a 57,600 baud serial connection to the uM-FPU configured as 8 data bits, no parity, and 1 stop bit. The connection is shown below.



For more information on ways to make the serial connection, see *Application Note 9 – Adding a Serial Connection to the uM-FPU V2*.

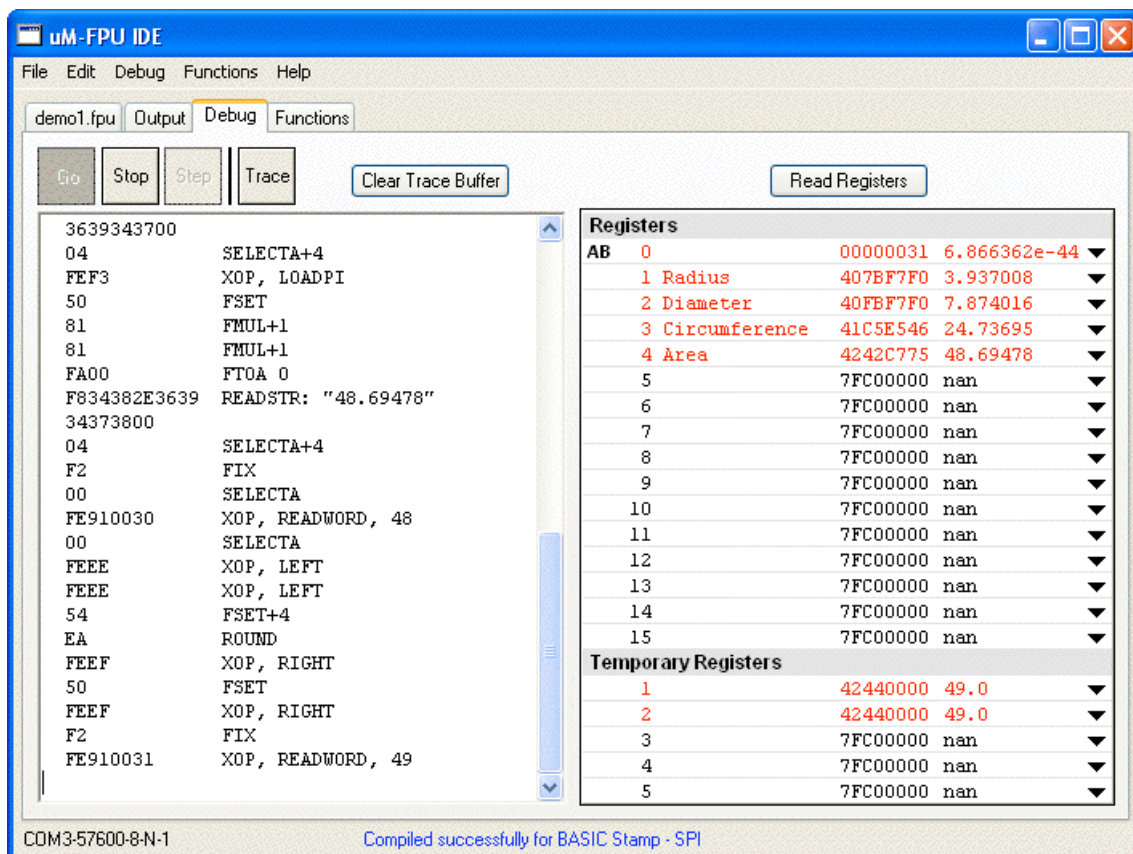
The **Trace on Reset** parameter determines whether debug tracing is enabled at Reset. Since tracing is disabled by default, we will select the **Set Parameters...** item from the **Functions** menu, and make sure that the **Trace on Reset** option is selected. Check that the **Set Parameters** dialog looks the same as the one shown below, then click OK. If you get an error message, check all of your connections, reset the uM-FPU, and try again. If the error persists, you may need to power the uM-FPU chip off and on to ensure a proper Reset.



Select the **Debug** window, and click the **Clear Trace Buffer** button.

Run the program that you developed in the previous tutorial. The trace buffer on the left should display the information shown below.

Click the **Read Registers** button. The register panel on the right should now display the information shown below.



Scroll up to the beginning of the trace buffer. You should see a Reset message similar to the following:

```
-----
RESET: 2004-08-07 13:19:31
-----
```



Every time the uM-FPU resets a reset message is displayed.

Compare the instructions in the debug trace to your program. You can see how useful tracing is for checking the actual sequence of instruction executed by the uM-FPU. Coding errors or logic errors can often be found simply by examining the trace.

To experiment with breakpoints and single stepping, add the following line to your program at a spot that you want a breakpoint to occur at.

```
SHIFTOUT FpuOut, FpuClk, MSBFIRST, [XOP, BREAK]
```

After adding the breakpoint has been added to your program, run the program again. You should now get a breakpoint.

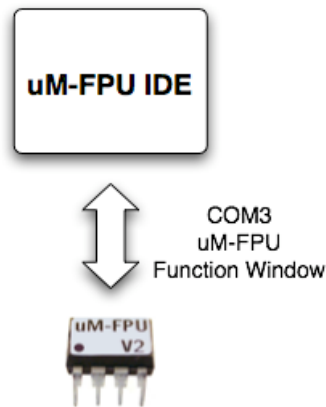
Click the  button to single step through the code, or the  button to continue execution. Check the section entitled *Reference Guide for Debugging* for more information.

This completes the tutorial on debugging uM-FPU code. Using the information gained from this tutorial, and more detailed information from the reference section, you should now be able to use the IDE to debug your own programs.

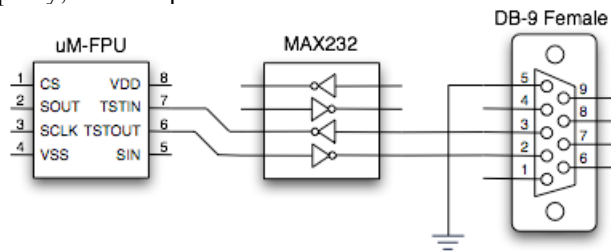
## Tutorial 3 - Storing User-Defined Functions

The figure below shows the process of storing user-defined functions. A serial connection (COM3 in this example) is required to connect the uM-FPU IDE to the built-in debugger on the uM-FPU chip. The uM-FPU debugger provides support for reading and programming the Flash memory used to store user-defined functions.

### Storing User-Defined Functions.



To store user-defined functions, the uM-FPU IDE requires a 57,600 baud serial connection to the uM-FPU configured as 8 data bits, no parity, and 1 stop bit. The connection is shown below.



For more information on ways to make the serial connection, see *Application Note 9 – Adding a Serial Connection to the uM-FPU V2*.

### Defining functions

A good way to develop user-defined functions is to first develop and test them as regular code. Once the code is working correctly, functions can be defined and stored on the uM-FPU chip. In the previous tutorials we developed and tested code to calculate the diameter, circumference, and area of a circle. We will now store these calculations as user-defined functions.

The `#FUNCTION` directive is used to define a function. It specifies the number of the function (0-63) and an optional name. Any code that appears after the `#FUNCTION` directive will be stored in the function. The end of a function occurs at the next `#FUNCTION` directive, an `#END` directive, or the end of the source file.

### Calling Functions

A function is called by entering an ampersand (&) before the function name or number.

e.g.

```
@GetDiameter
```

### Modifying the Code for Functions

Open the source file called *tutorial1.fpu* that you previously saved. Add a `#FUNCTION` directive before the diameter, circumference and area calculations, and add an `#END` directive after the area calculation. Move the radius

calculation to after the function definitions, and add a call to the three functions. The source code will now look as follows:

```

distance      VAR  Word
areaIn        VAR  Word

Radius        equ  F1
Diameter      equ  F2
Circumference equ  F3
Area          equ  F4

#FUNCTION 1 GetDiameter
Diameter = Radius * 2

#FUNCTION 2 GetCircumference
Circumference = PI * Diameter

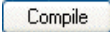
#FUNCTION 3 GetArea
Area = PI * Radius * Radius

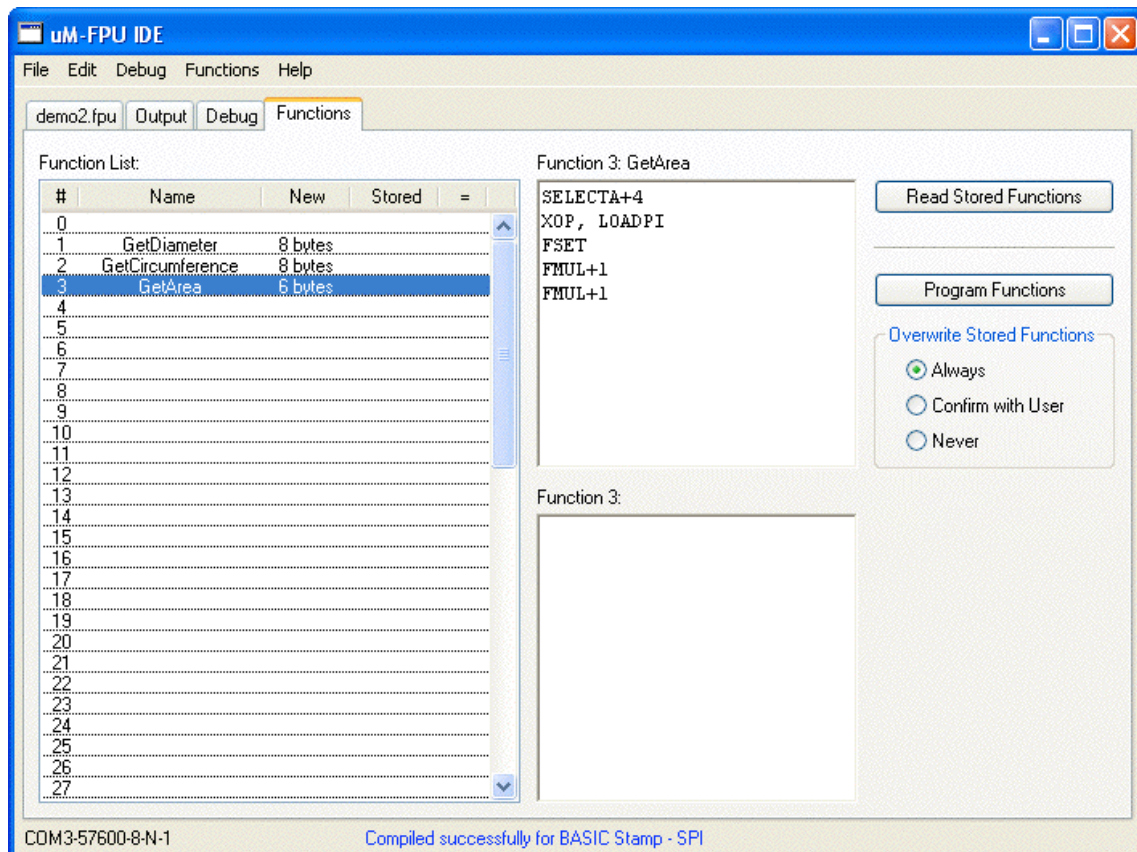
#END

Radius = distance / 1000
@GetDiameter
@GetCircumference
@GetArea

areaIn = ROUND(area)
    
```

### Compile and Review the Functions

Click the  button, then select the Functions tab. The following window should be displayed:

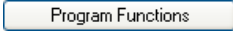




The figure above shows that three functions have been defined. The code for each function is displayed in the upper middle panel. Click on the other functions in the function list to see the code associated with them.

### Storing the Functions

Make sure that the **Overwrite Stored Functions** preference is set to **Always** (as shown in the figure above).

Click the  button to store the user-defined functions on the uM-FPU chip. A status dialog will be displayed as the uM-FPU functions are programmed into Flash memory. If an error occurs, check the connection, turn the power to the uM-FPU on and off to ensure that it is reset, and try again.

### Running the Program

Copy the generated code from the uM-FPU IDE to the BASIC Stamp program, and replace the diameter, circumference and area calculations with function calls. The main routine in your BASIC Stamp program should now look like the following:

```

Main:
  distance = 2575

  '--- Radius = distance / 1000
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [Radius,
    LOADWORD, distance.HIGHBYTE, distance.LOWBYTE, FSET,
    FWRITEB, $44, $7A, $00, $00, FDIV]
  DEBUG CR, "Radius:          "
  format = 63
  GOSUB Print_FloatFormat

  '--- Diameter = Radius * 2
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [XOP, FUNCTION+1]
  DEBUG CR, "Diameter:        "
  format = 63
  GOSUB Print_FloatFormat

  '--- Circumference = PI * Diameter
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [XOP, FUNCTION+2]
  DEBUG CR, "Circumference:    "
  format = 63
  GOSUB Print_FloatFormat

  '--- Area = PI * Radius * Radius
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [XOP, FUNCTION+3]
  DEBUG CR, "Area:              "
  format = 63
  GOSUB Print_FloatFormat

  '--- areaIn = Area
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [Area]
  GOSUB Fpu_Wait
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FIX, SELECTA]
  GOSUB Fpu_Wait
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [XOP, READWORD]
  SHIF TIN FpuIn, FpuClk, MSBP RE, [areaIn.HIGHBYTE, areaIn.LOWBYTE]
  DEBUG CR, "AreaIn:          ", DEC AreaIn

END

```

Save the source file as *Tutorial2.fpu* and save the BASIC Stamp program *Tutorial2.bs2*, then run the program. The following output should be displayed in the terminal window:

```

uM-FPU V2.0
Radius:      2.575
Diameter:    5.150
Circumference: 16.179
Area:        20.831
AreaIn :     21

```

Note: If the user-defined functions have not been stored properly, the output will look like the following:

```

uM-FPU V2.0
Radius:      2.575
Diameter:    2.575
Circumference: 2.575
Area:        2.575
AreaIn:      0

```

Calling an undefined functions has no effect, so the A register remains unchanged after the Radius calculation, and the same value prints out for each `Print_Format` call.

This completes the tutorial on storing user-defined functions. Using the information gained from this tutorial, and the more detailed information in the reference section, you should now be able to define and store your own functions with the IDE.

## Reference Guide for Generating uM-FPU Code

The source file is used to enter symbol definitions and math expressions to implement floating point or long integer calculations. Expression can contain uM-FPU registers, microcontroller variables, constants, math operators, math functions and parentheses.

### Order of Evaluation

Expressions evaluate from left to right with no operator precedence.

$$F1 = F2 + F3 * F4$$

would result in F1 being set to the value of F2 added to F3 then multiply by F4.

Parentheses are used to change the order of operations. For example,

$$F1 = F2 + (F3 * F4)$$

would result in F1 being set to the value of F2 added to the value of F3 multiplied by F4.

Multiple constant values entered one after another are automatically reduced to a single constant value in the expression. For example,

$$F1 = F2 * 5 / 2$$

would result in F1 being set to the value F2 multiplied by 2.5.

If you don't want constants to be reduced, you need to use parentheses. The familiar expression for converting temperature from Celsius to Fahrenheit would be entered as:

$$F1 = (F2 * 9 / 5) + 32$$

If no parentheses were used, the expression would be calculated as F2 multiplied by 33.8, which would be incorrect.

The code generator often adds one level of parenthesis, so parentheses in expressions should only be nested up to four levels deep (including the parentheses used for functions).

### Pre-defined Register Names

The following register definitions pre-defined:

F0, F1, F2, ... F15	specifies that register 0 to 15 contains a floating point value
L0, L1, L2, ... L15	specifies that register 0 to 15 contains a long integer value
U0, U1, U2, ... U15	specifies that register 0 to 15 contains an unsigned long integer value

### Pre-defined Constants

PI	constant value for pi (3.1515926)
E	constant value for e (2.7182818)

### Math Operators

+	Plus
-	Minus
*	Multiply
/	Divide

### Math Functions

SQRT, LOG, LOG10, EXP, EXP10, SIN, COS, TAN, FLOOR, CEIL, ROUND, NEGATE, ABS, INV, DEGREES, RADIANS, FLOAT, FIX, COMPARE, STATUS, POWER, ROOT, MIN, MAX, FRAC, ASIN, ACOS, ATAN, ATAN2, LCOMPARE, ULCOMPARE, LSTATUS, LNEGATE, LABS

### User-defined Names

Using symbolic names can make expressions much easier to read and understand. Symbol names can be defined for uM-FPU registers, microcontroller variables, and constants.

### uM-FPU Registers

Registers are defined using the EQU operator to assign a new name to a previous register definition.

e.g.

```
Y      EQU  F1
X      EQU  F2
Radius EQU  F1
```

### Constants

Constants are defined using the CON or EQU operator.

e.g.

```
Length CON  4.75
```

or

```
Length EQU  4.75
```

The compiler simplifies constant expressions to a single constant value.

e.g.

```
Pi2     CON  PI / 2
```

### Microcontroller Variables

Microcontroller variables are defined using the VAR or EQU operator and one of the following keywords:

```
BYTE      8-bit signed integer value
UBYTE     8-bit unsigned integer value
WORD      16-bit signed integer value
UWORD     16-bit unsigned integer value
LONG      32-bit signed integer value
ULONG     32-bit unsigned integer value
FLOAT     32-bit floating point value
```

e.g.

```
count      EQU  BYTE
sensorInput EQU  UWORD
lastAngle  EQU  FLOAT
```

When a microcontroller variables is used in an expression the uM-FPU compiler generates the necessary code to transfer the value between the microcontroller and the uM-FPU.

e.g.

```
degreesC EQU  BYTE
degreesF EQU  F1
degreesF = (degreesC * 9 / 5) + 32
```

The generated code will load the byte value from the microcontroller, convert it to floating point, multiply by 1.8 and then add 32.

### Comments

Comments can be added to the end of a line by entering either an apostrophe ( ' ), double slash ( // ) or semicolon ( ; ).

### User-defined Functions

User-defined functions are specified using the #FUNCTION directive. After a #FUNCTION directive is encountered, all compiled code is stored in the function specified. The end of a function occurs at the next #FUNCTION directive, #END directive, or the end of the source file. The #FUNCTION directive can optionally include a function name that can be used in the remainder of the source file to call the function.

e.g.

```
#FUNCTION 1 GetDiameter
```

A function call is specified by using the @ character followed by a constant value between 0 and 63 representing the number of the function to call.

e.g.

```
@1      ' call function 1
```

Or, by using the the @ character followed by the name of a previously defined function.

e.g.

```
@AddValue      ' call function AddValue
```

An example of a function definition and function call is as follows:

```
Value1 EQU BYTE
Value2 EQU BYTE
X      EQU F1
Y      EQU F2
Z      EQU F3

#FUNCTION 1 Hypotenuse
Z = SQRT(X*X + Y*Y)
#END

X = Value1
Y = Value2
@Hypotenuse
```

If a function is called from inside another function, execution will not return to the original function (i.e. it is not a subroutine call). This can still be used to chain functions together to create larger functions (since the maximum size of a single function is 256 bytes), or for sequence of operations. For example, if you were updating the position of a robotic arm, you could chain through relative offsets of each joint to get the cumulative offset.

e.g.

```
#FUNCTION 1 AddShoulder
X = X + ShoulderX
Y = Y + ShoulderY
Z = Z + ShoulderZ

#FUNCTION 2 AddElbow
X = X + ElbowX
Y = Y + ElbowY
Z = Z + ElbowZ
@AddShoulder

#FUNCTION 3 AddWrist
X = X + WristX
Y = Y + WristY
Z = Z + WristZ
@AddElbow
#END
```

### Entering uM-FPU Assembler Code

The expression compiler takes regular math expressions and compiles them to produce the required uM-FPU instructions. There are some capabilities of the uM-FPU that are not easily accessible using math expressions, or in some cases it may be desirable to write a more optimized version of the uM-FPU code. The #ASM and #ENDASM directives are used to specify uM-FPU instructions ( assembler code). The syntax of the assembler code instructions is shown below. Multiple instructions can be entered on a single line, and an instruction can span more than one line, but each element of an instruction (e.g. a number or string) must be on a single line.

e.g.

```
#ASM SELECTA+1 XOP LOADPI FSET #ENDASM
```

or

```
#ASM
SELECTA+1
XOP LOADPI
FSET
#ENDASM
```

**Assembler Instructions**

SELECTA+r	LOADBYTE bb	XOP LWRITEA zzzz
SELECTB+r	LOADUBYTE bb	XOP LWRITEB zzzz
FWRITEA+r yyyy	LOADWORD wwww	XOP LREAD
FWRITEB+r yyyy	LOADUWORD wwww	XOP LUDIV
FREAD+r	READSTR	XOP POWER
FSET+r	ATOF "ssss"	XOP ROOT
LSET+r	FTOA "ssss"	XOP MIN
FADD+r	ATOL ff	XOP MAX
FSUB+r	LTOA ff	XOP FRACTION
FMUL+r	FSTATUS	XOP ASIN
FDIV+r	NOP	XOP ACOS
LADD+r	XOP FUNCTION+n	XOP ATAN
LSUB+r	XOP IF_FSTATUSA	XOP ATAN2
LMUL+r	XOP IF_FSTATUSB	XOP LCOMPARE
LDIV+r	XOP IF_FCOMPARE	XOP LUCOMPARE
	XOP IF_LSTATUSA	XOP LSTATUS
SQRT	XOP IF_LSTATUSB	XOP LNEGATE
LOG	XOP IF_LCOMPARE	XOP LABS
LOG10	XOP IF_LUCOMPARE	XOP LEFT
EXP	XOP IF_LTST	XOP RIGHT
EXP10	XOP TABLE	XOP LOADZERO
SIN	XOP POLY	XOP LOADONE
COS	XOP READBYTE	XOP LOADE
TAN	XOP READWORD	XOP LOADPI
FLOOR	XOP READLONG	XOP LONGBYTE bb
CEIL	XOP READFLOAT	XOP LONGUBYTE bb
ROUND	XOP LINCA	XOP LONGWORD wwww
NEGATE	XOP LINCB	XOP LONGUWORD www
ABS	XOP LDECA	XOP IEEEEMODE
INVERSE	XOP LDECB	XOP PICMODE
DEGREES	XOP LAND	XOP CHECKSUM
RADIANS	XOP LOR	XOP BREAK
SYNC	XOP LXOR	XOP TRACEOFF
FLOAT	XOP LNOT	XOP TRACEON
FIX	XOP LTST	XOP TRACESTR "ssss"
FCOMPARE	XOP LSHIFT	XOP VERSION

r	register number (0-15)
n	function number (0-63)
bb	8-bit value
wwww	16-bit value
yyyy	floating point value
zzzz	long integer value
ssss	ASCII string

Note: Extended opcode instructions are required to have an XOP before the opcode.

**Assembler Directives**

#BYTE bb	8-bit byte value
#WORD wwww	16-bit word value
#LONG zzzz	long integer value
#FLOAT yyyy	floating point value

**Wait Code**

The uM-FPU has a 32 byte instruction buffer. If a sequence of instructions in a calculation exceeds 32 the buffer could overflow, so the program must wait for the buffer to empty at least every 32 bytes. The code generated by the IDE accounts for this, and will insert a wait sequence as required.

## File Menu

File	
New...	Ctrl+N
Open...	Ctrl+O
Save	Ctrl+S
Save As...	Ctrl+Shift+S
<hr/>	
Exit	Ctrl+Q

The **New...** menu item creates a new source file and sets the name to *untitled*. If a previous source file is open and has been changed since the last time it was saved, the user will first be prompted to save the previous source file.

The **Open...** menu item opens an existing source file. A file open dialog will be displayed. If a previous source file is open and has been changed since the last time it was saved, the user will first be prompted to save the previous source file.

The **Save** menu item saved the source file. If the source file has not been previously saved, a file save dialog will be displayed.

The **Save As...** menu item displays a file save dialog.

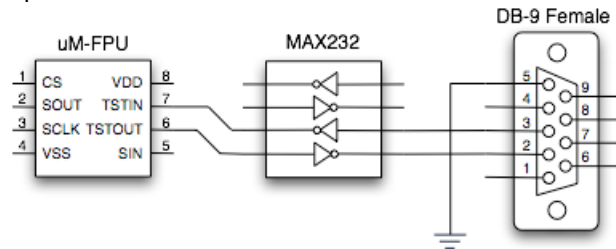
The **Exit** menu item causes the uM-FPU IDE to quit. If a source file is open and has been changed since the last time it was saved, the user will first be prompted to save the source file.

## Reference Guide for Debugging

Utilizing the built-in uM-FPU debugger, the IDE provides a high-level interface for debugging programs that use the uM-FPU floating point coprocessor. It supports the ability to trace uM-FPU instructions, set breakpoints, single-step through execution of uM-FPU instructions, and display the value of uM-FPU registers. The IDE includes a disassembler so that instruction traces are displayed in easy-to-read assembler format.

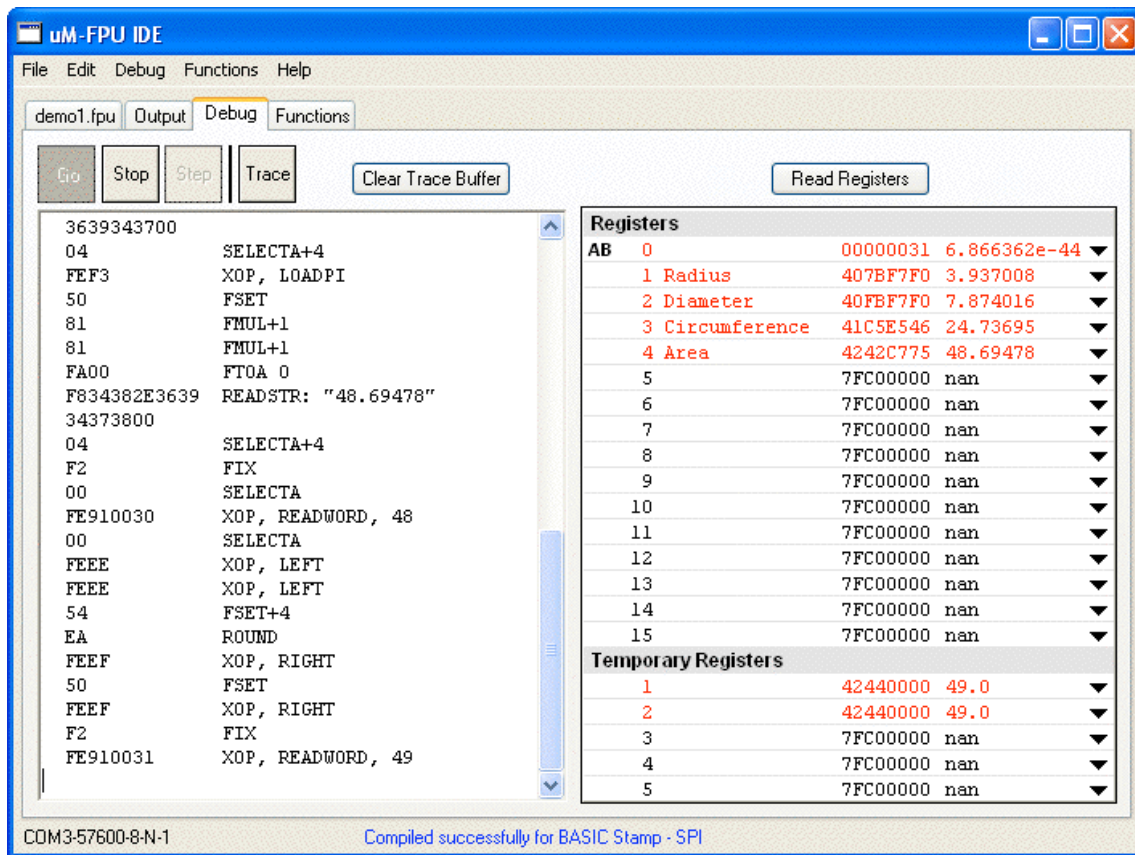
### Connecting the Debugger

To use the debugger, the uM-FPU IDE requires a 57,600 baud serial connection to the uM-FPU configured as 8 data bits, no parity, and 1 stop bit. The connection is shown below.



For more information on ways to make the serial connection, see *Application Note 9 – Adding a Serial Connection to the uM-FPU V2*.

An example of the Debug Window is shown below:





The scrolling window on the left displays trace messages, and the panel on the right displays the contents of the uM-FPU registers. The **Go/Stop/Stop/Trace** buttons at the top left control the breakpoint and trace features, and the connection status is displayed at the lower left of the window. Use the **Select Port...** menu item in the **Debug** menu if the port needs to be changed


### Trace Buffer

The scrolling window on the left of the debug window displays the trace buffer. When a Reset occurs a message is displayed showing the date and time of the Reset.

e.g.


```
-----
RESET: 2004-08-07 13:19:31
-----
```

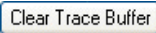
Tracing is turned off at Reset, unless the **Trace on Reset** parameter has been set. Tracing can be controlled by

the program using the **TRACEON** and **TRACEOFF** instructions, or by the user with the  button. If tracing is enabled, all uM-FPU instructions are displayed as they are executed. The opcode and data bytes are displayed on the left, and the uM-FPU instructions are displayed on the right in assembler format.


e.g.

```
TRACE: ON
04          SELECTA+4
FEF3       XOP, LOADPI
50          FSET
81          FMUL+1
81          FMUL+1
FA00       FTOA 0
F834382E3639 READSTR: "48.69478"
34373800
...
```

The  button toggles the trace mode on and off.




Clicking the  button will clear the contents of the trace buffer.

### Breakpoints

Breakpoints can be inserted into a program using the **BREAK** instruction, or initiated by the user with the  button. Breakpoints occur after the next uM-FPU instruction finishes executing (except for **SELECTA** or **SELECTB**). When a breakpoint occurs, the last uM-FPU instruction executed before the breakpoint is displayed, followed by the break message.

e.g.

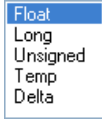
```
04          SELECTA+4
FEF3       XOP, LOADPI
BREAK
```


The    buttons are enabled or disabled depending on the current state of execution. The **Go** button is used to continue execution, and is enabled at Reset or after a breakpoint occurs. The **Stop** button is used to stop execution after the next uM-FPU instruction is executed. If the uM-FPU is idle when the **Stop** button is pressed, the breakpoint will not occur until the next uM-FPU instruction is executed. If the uM-FPU is already at a breakpoint, then the **Stop** button will be disabled. The **Step** button is used to single step through instructions. A new breakpoint occurs after each instruction (except for **SELECTA** or **SELECTB**).

### The Register Panel

The register panel displays the value of each register and indicates the currently selected A register and B register. The A and B registers are indicated by an A and B marker in the left margin of the register panel. For each register, the register number, optional register name, hexadecimal value, and formatted value is displayed. The formatted value can be displayed as floating point, long integer, or unsigned long integer. Clicking the small triangle on the right ▼ displays a pop-up menu that is used to select the display format or name of the register to display (if names have been assigned).

e.g.



The current register values are automatically updated after every breakpoint. The  button can be used to manually force an update of the register values. Register values are displayed in red if the value has changed since the last time the display was updated, or black if the value is unchanged.

### Register Names

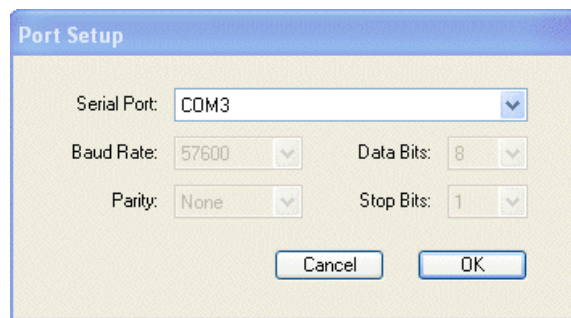
Register names are automatically set from the register definitions in the source file. There can be multiple definitions for each register. To select the name to display, click the small triangle on the right ▼ and select the name from the pop-up menu. The register display format will also be set to the format specified in the register definition.

## Debug Menu



The **Select Port...** menu item is used to select the serial communications port. The following dialog will be displayed.

### Port Setup Dialog



The **Go**, **Stop**, and **Step** menu items have the same function as the **Go**, **Stop** and **Step** buttons.

The **Turn Trace On / Turn Trace Off** menu item has the same function as the **Trace** button.

The **Trace on Reset** menu item will automatically enable tracing after a **Reset**.

The **Read Registers** menu item has the same function as the **Read Registers** button.

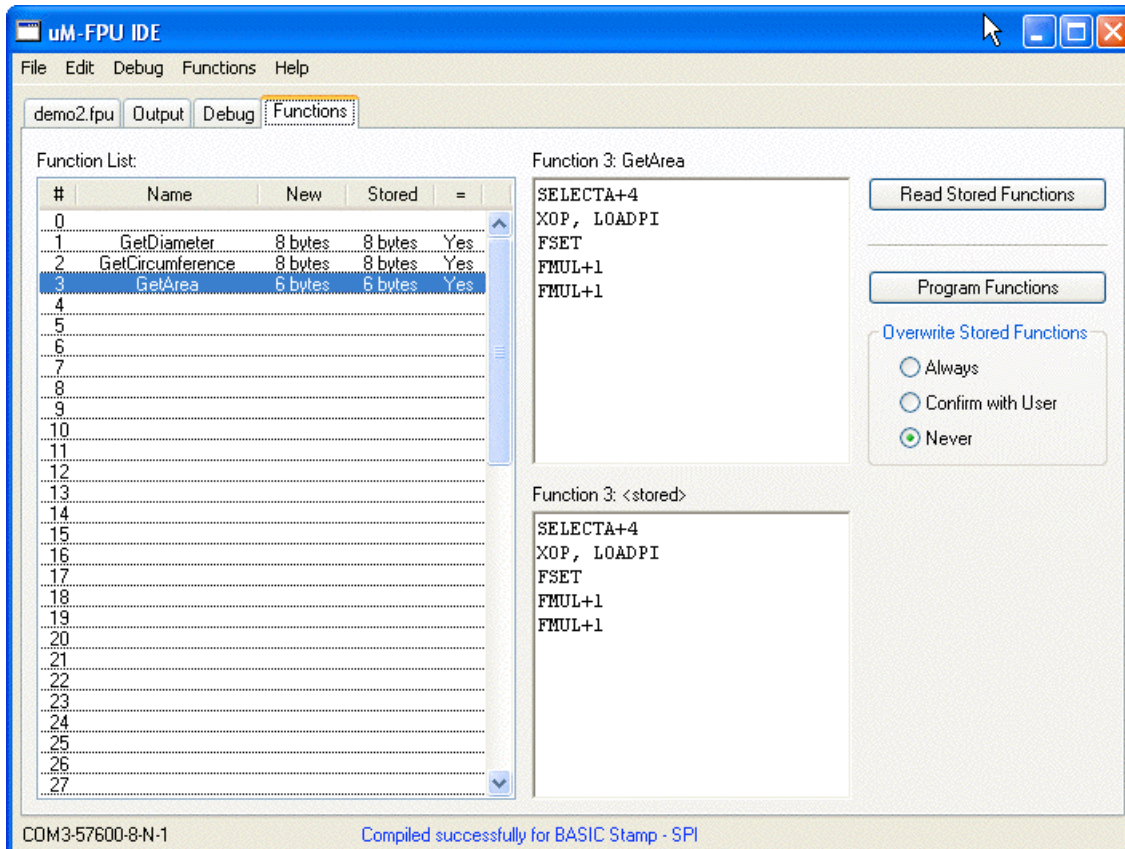
The **Read Version** menu item will display the version of the uM-FPU in the trace window.

The **Read Checksum** menu item will display the checksum of the uM-FPU in the trace window.

## Reference Guide for Storing User-Defined Functions


The Functions window provides support for storing user-defined functions on the uM-FPU chip. Using stored functions you can reduce memory usage on the microcontroller, simplify the interface and often increase the speed of operation. The uM-FPU reserves 1024 bytes of flash memory for storing functions and parameters. Functions are stored as a string of uM-FPU instructions, and up to 64 functions can be defined. Functions are specified in the source file by using the #FUNCTION directive (details of the #FUNCTION directive are provided in the section entitled *Reference Guide for Generating uM-FPU Code*).


### Function Window

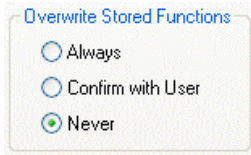


The scrolling list on the left shows all of the currently defined functions. The **Name** column displays the name of the new function if it is defined in the source file. The **New** column shows the size in bytes of the new functions defined in the source file. The **Stored** column displays the size in bytes of the functions currently stored on the uM-FPU chip. The **=** column displays **Yes** if the new and stored functions are the same, or **No** if they are different.

The panel located at top center displays the uM-FPU instructions for the new function defined in the source file, and the panel located at bottom center displays the uM-FPU instructions for the function stored on the uM-FPU chip. The function to be displayed is selected by clicking on one of the functions in the Function List.

Clicking the  button reads all of the functions currently stored on the uM-FPU chip and updates the function list.

Clicking the  button programs the uM-FPU Flash memory to store all of the functions defined in the function list. If a newly defined function is different than the currently stored functions, the action taken is determined by the **Overwrite Stored Functions** option.

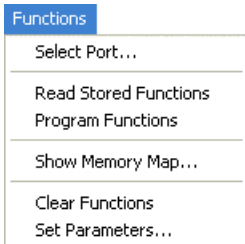


If the **Always** option is selected, a new function will always overwrite any previously stored function.

If the **Confirm with User** option is selected, the user is asked to confirm whether a new function should replace the previously stored function.

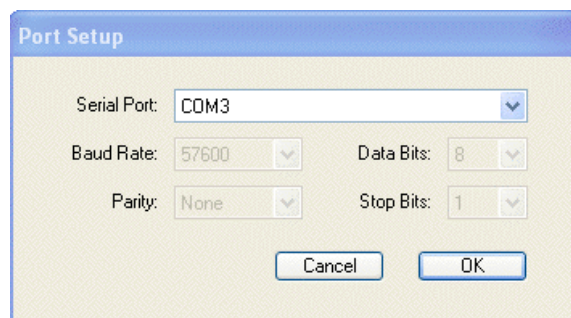
If the **Never** option is selected, new function are not allowed to replace previously stored functions.

## Functions Menu



The **Select Port...** menu item is used to select the serial communications port. The following dialog will be displayed.

### Port Setup Dialog

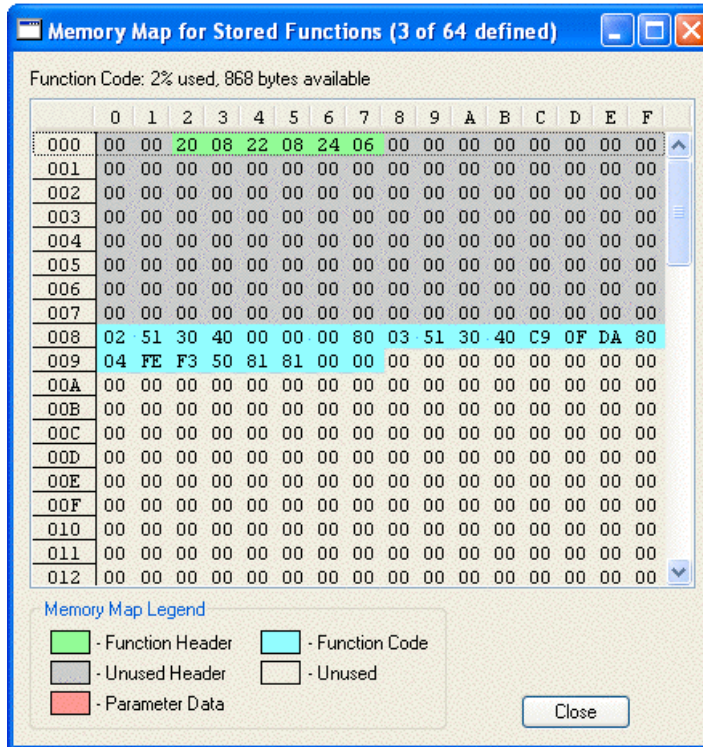


The **Read Stored Functions** menu item has the same function as the **Read Stored Functions** button.

The **Program Functions** menu item has the same function as the **Program Functions** button.

The **Show Memory Map...** menu item displays a memory map showing the usage of the uM-FPU Flash memory area reserved for user-defined functions. A status line at the top shows the percent of memory used and the number of bytes available.

### Memory Map Dialog



The **Clear Functions** menu item will clear all of the user-defined functions from the uM-FPU. A dialog will be displayed requesting confirmation before the functions are cleared from memory.

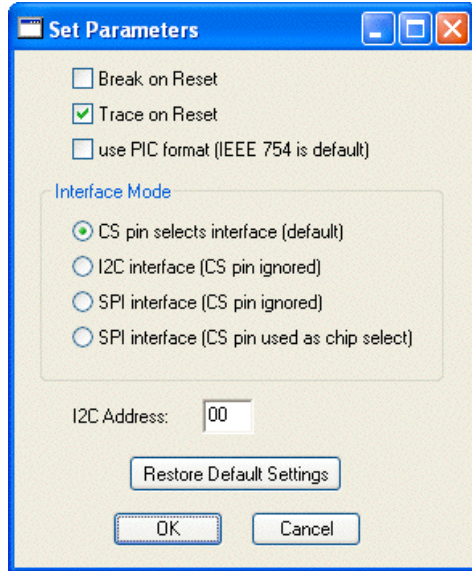
The **Program Functions** menu item has the same function as the **Program Functions** button.

The **Set Parameters...** menu item is used to set the uM-FPU parameters as described in the next section.

## Reference Guide for Setting uM-FPU Parameters

The Set Parameters... menu item is used to set the uM-FPU parameters.

### Set Parameters Dialog



#### Break on Reset

If this option is selected, a breakpoint will occur on the first instruction following a Reset.

#### Trace on Reset

If this option is selected, debug tracing is turned on at Reset.

#### use PIC Format (IEEE 754 is default)

If this option is selected, the PIC format will be used for reading and writing floating point values. Internally, the uM-FPU uses floating point values that conform to the IEEE 754 32-bit floating point standard. This is also the default format for reading and writing floating point values in uM-FPU instructions. The alternate PIC format is often used by PICmicro compilers. If this option is selected, floating point values are automatically translated between the PIC format and the IEEE 754 format whenever values are read from the uM-FPU or written to the uM-FPU, and the microcontroller program can use the PIC format. The `IEEEEMODE` and `PICMODE` instructions can be used to dynamically change the format. For additional information regarding the `IEEEEMODE` and `PICMODE` instructions, see the *uM-FPU V2 Instruction Reference*.

Note: The IDE code generator currently only generates code for the default IEEE 754 format. If the PIC format is used you would need to fix the data values in the code generated for `FWRITEA` and `FWRITEB` instructions.

#### Interface Mode

By default, the CS pin on the uM-FPU selects the SPI or I<sup>2</sup>C interface. The interface mode parameter can be used to select SPI or I<sup>2</sup>C at Reset (ignoring the CS pin), or it can be set to SPI mode with the CS pin acting as a chip select.

Note: Using SPI mode with the CS pin acting as a chip select is intended for special applications. All of the SPI support software currently supplied by Micromega assumes that no chip select is used. If this option is enabled, the user must develop special code to take advantage of the capability.

### I<sup>2</sup>C Address

By default, the I<sup>2</sup>C address used by the uM-FPU is 1100100x (binary), or C8 (hexadecimal). If the default address conflicts with another I<sup>2</sup>C device, or if multiple uM-FPU chips are used on the same I<sup>2</sup>C bus, the address can be changed to any other valid I<sup>2</sup>C address. The address is entered as an 8-bit hexadecimal number (with the lower bit ignored). A value of 00 will select the default C8 address.

### Restore Default Settings

The parameters are restored to the following default settings:

- No Break on Reset
- No Trace on Reset
- IEEE 754 mode
- CS pin selects I<sup>2</sup>C or SPI
- I<sup>2</sup>C address is C8

### 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
- Application Note 9 – Adding a Serial Connection to the uM-FPU V2*

Check the Micromega website at [www.micromegacorp.com](http://www.micromegacorp.com) for up-to-date information.