Micromega Corporation

Using the uM-FPU V3 Integrated Development Environment (IDE)

Introduction

The uM-FPU V3 Integrated Development Environment (IDE) software provides a set of easy-to-use tools for developing applications using the uM-FPU V3 floating point coprocessor. The IDE runs on Windows 98, NT, ME, 2000 and XP, and provides support for the following tasks:

- Generating uM-FPU V3 Code
- Debugging uM-FPU V3 Code
- Storing User-Defined Functions and Setting Parameters

Generating uM-FPU V3 Code

The Code Generator takes a source file containing definitions and math expressions, compiles it, and generates uM-FPU V3 code for the 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. Using copy-and-paste, the generated code can be easily copied into the main program for your microcontroller.

The IDE has a built-in editor for entering definitions and math expressions in the source file. The source file is stored as a text file with a default filename extension of `.fpu`.

Debugging uM-FPU V3 Code

The Debugger interacts with the built-in debug monitor on the uM-FPU V3 chip to display instruction traces and the contents of internal registers. It also allows you to set breakpoints and step through the execution of uM-FPU instructions.

Storing User-Defined Functions and Setting Parameters

The Function Programmer stores user-defined functions and parameters in Flash memory on the uM-FPU V3 chip. Stored functions can reduce memory usage on the microcontroller, simplify the interface, and increase the speed of execution.

A serial connection is required between the PC running the IDE and the uM-FPU V3 chip for debugging and storing user-defined functions. Code generation can be done without a serial connection. There are various ways of providing a serial connection including: RS-232 adapters, Serial-to-USB adapters, or adding a level converter such as the MAX-232.

This document contains the following sections:

- Overview of uM-FPU V3 IDE User Interface
- Tutorial 1: Generating uM-FPU V3 Code
- Tutorial 2: Debugging uM-FPU V3 Code
- Tutorial 3: Storing User-Defined Functions
- Reference Guide: Generating uM-FPU V3 Code
- Reference Guide: Debugging uM-FPU V3 Code
- Reference Guide: Storing User-Defined Functions
- Reference Guide: Setting uM-FPU V3 Parameters

The tutorials demonstrate how to use the IDE by going through some simple examples. The examples use the BASIC Stamp with a SPI interface as the target. If you work with a different microcontroller or compiler, the procedures are the same, but the output code for the selected target will be different. The reference guides provide more detailed information about the features of the IDE.

Overview of uM-FPU V3 IDE User Interface

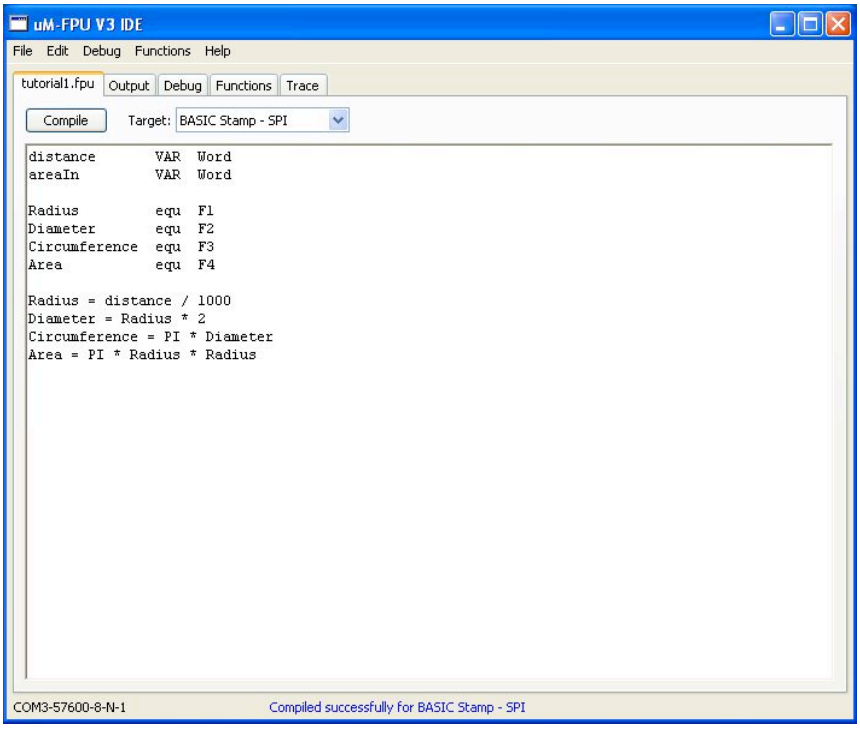
The main window of the IDE has tabs to select one of the following window panes:

- Source File
- Output
- Debug
- Functions
- Trace

Clicking the tab will display the associated window pane. 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 Source File, Output, Debug and Function windows are shown below. The Trace window shows a trace of the serial data exchanged between the IDE and the uM-FPU V3 chip. It's provided for reference purposes and is not normally used.

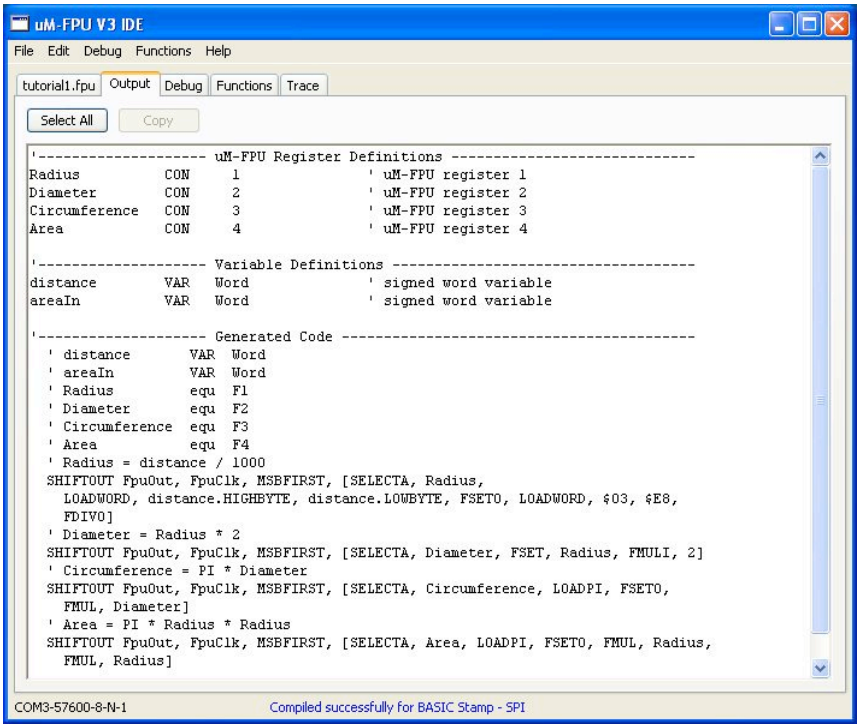
Source File Window

Used to display and edit the source file.



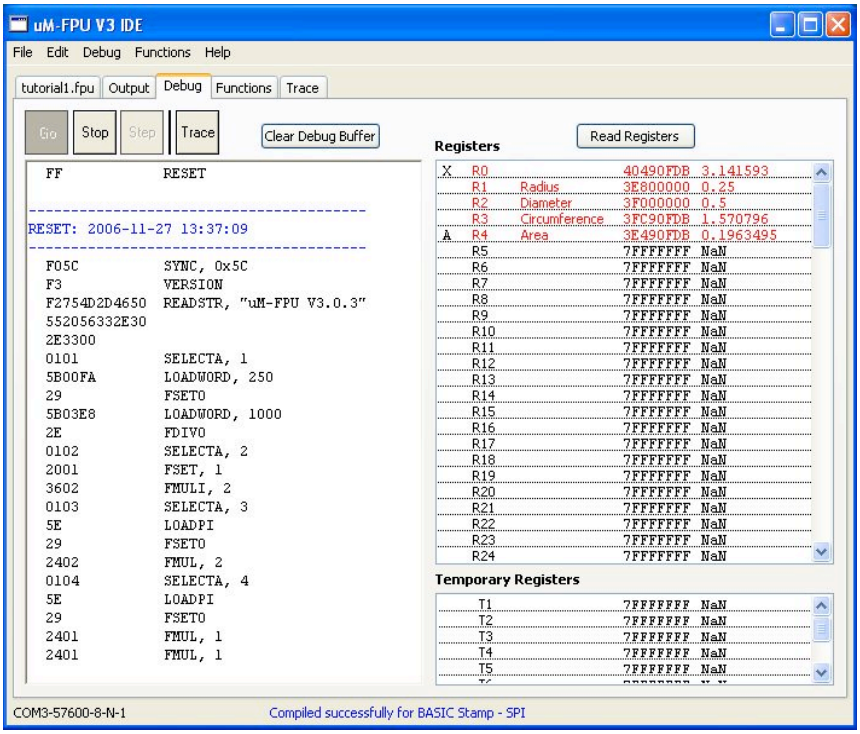
Output Window

Used to display the generated uM-FPU V3 code.



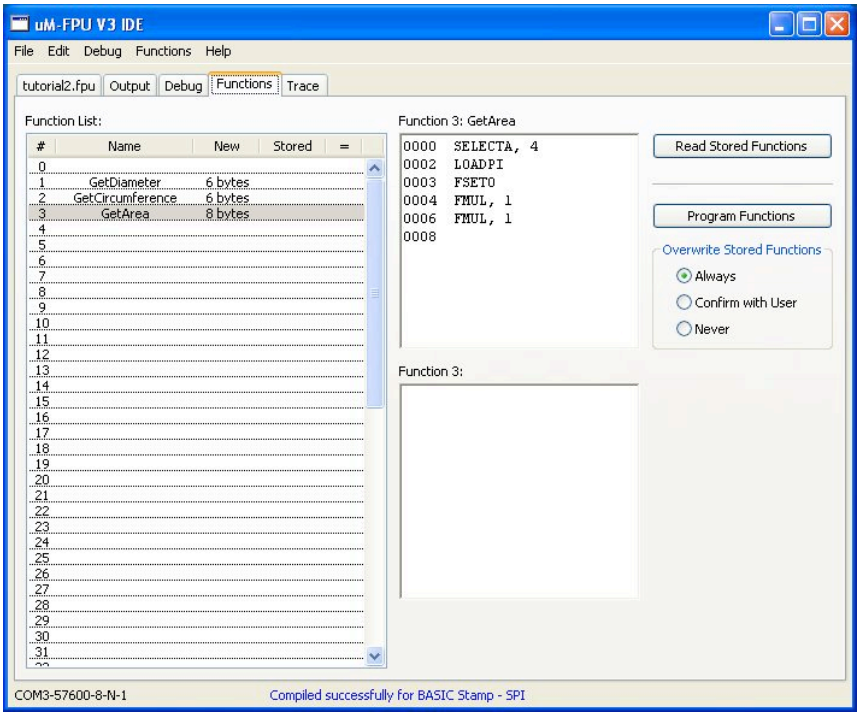
Debug Window

Used for debugging uM-FPU V3 code.



Functions Window

Used to store user-defined functions.

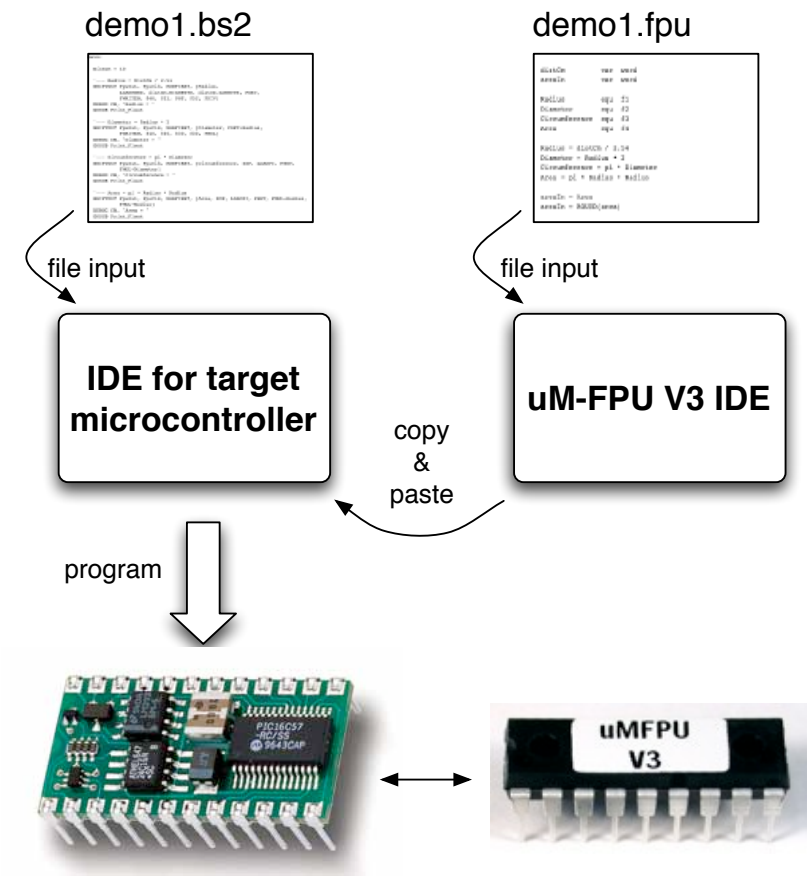


Tutorial 1: Generating uM-FPU V3 Code

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: Generating uM-FPU V3 Code*.

The figure below shows the process of developing uM-FPU V3 code using the IDE. The left side of the diagram shows the normal steps for developing microcontroller code. Source code is entered into a program file (called *demo1.bs2* in this example) and compiled using the development environment for the selected target. The right side of the diagram shows the additional steps for generating uM-FPU V3 code. Definitions and math expressions are entered into a source file (called *demo1.fpu* in this example), the file is compiled using the uM-FPU V3 IDE, and using copy-and-paste, the generated code is copied from the IDE to the main program file. The main program file is then compiled and run as usual using the development environment for the selected target.

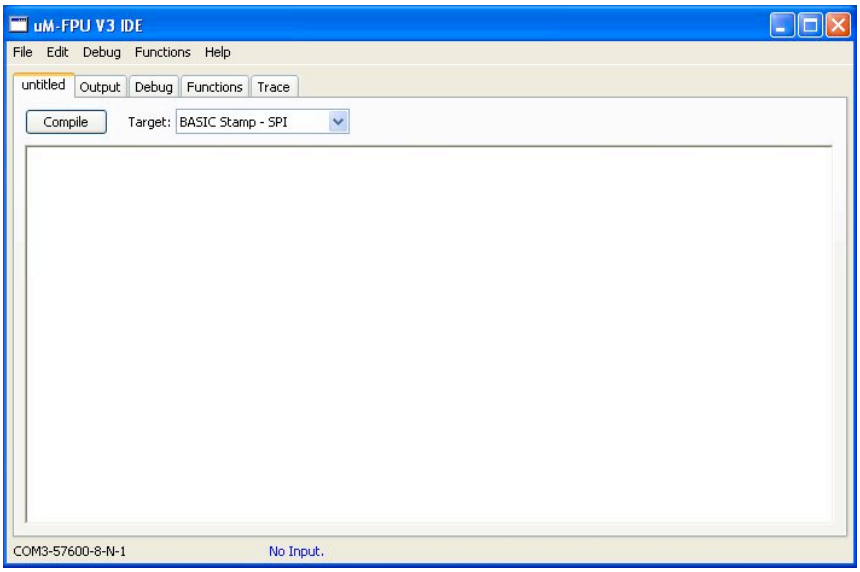
Developing uM-FPU V3 code



Starting the uM-FPU V3 IDE

Start the uM-FPU V3 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 as the target for 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
At start of program.

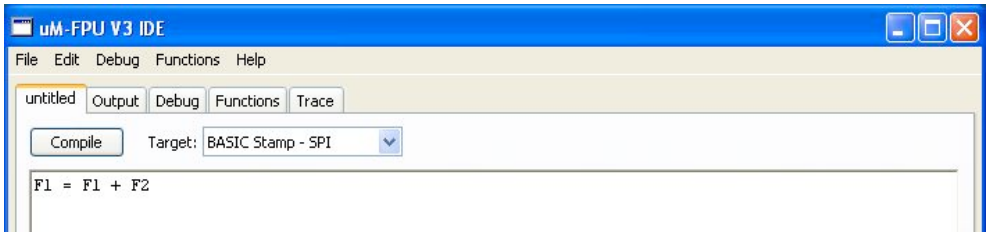


A Quick Introduction to Generating Code

The uM-FPU V3 IDE uses predefined names for the 128 registers in the uM-FPU V3 chip. The pre-defined names `F0, F1, F2, … F127` are used to specify registers 0 through 127 and define the type as floating point. For example, to add the floating point values in register 1 and register 2, and store the result in register 1, you would enter the following expression:

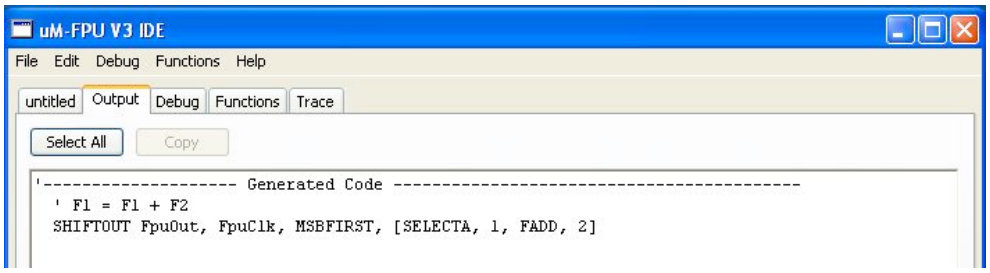
```
F1 = F1 + F2
```

The source file window should look as follows:



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  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 register A, then add the value of register 2 to register A. You've successfully compiled your first expression. (If you want to see the code generated for a different target, go back to the source file window and select a different target from the Target pop-up menu.)

Defining Names

Math expressions can be easier to read when meaningful names are used. The IDE allows you to define names for uM-FPU registers, microcontroller variables and constants.

Registers are defined using the `EQU` operator and one of the predefined register names. Microcontroller variables are defined using the `VAR` operator. For example, the following statements define `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
```

Sample Project

Suppose we have a distance measuring device that returns a number of pulses proportional to distance. It measures 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, then calculate the diameter, circumference and area using the uM-FPU V3 chip. The results are displayed in units of inches to three decimal places.

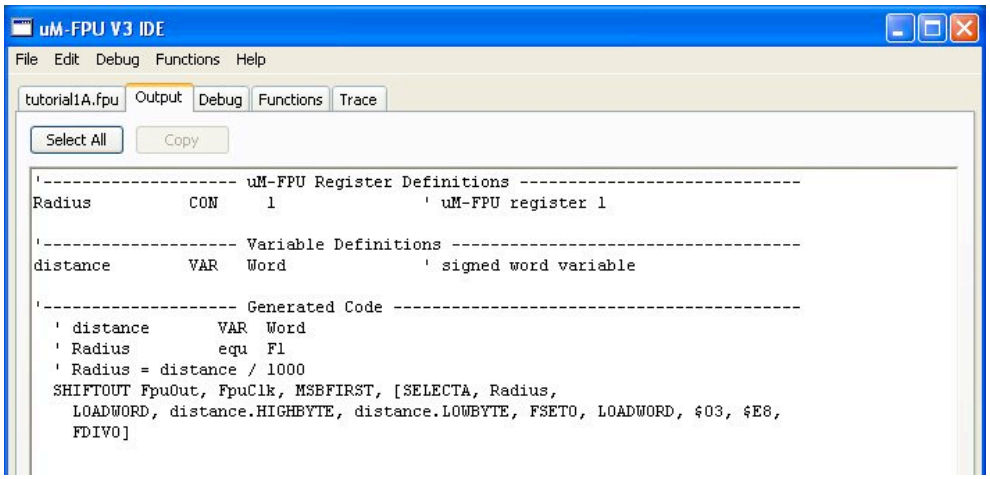
Calculating Radius

The number of pulses returned by the distance measuring device ranges 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, we'll divide the distance value by 1000 once it's loaded to the uM-FPU V3 chip.

Clear the source file (you can use the File/New... menu item), and enter the following code:

```
distance  VAR  word
Radius    EQU  F1

Radius = distance / 1000
```

Click the Compile button, then click the Output tab. The generated code should look as follows:



The generated code does the following:

- the `Radius` register is selected as register A
- the `LOADWORD` instruction takes the 16-bit value of `distance` from the microcontroller, converts it to floating point and stored it in register 0
- the `FSET0` instruction sets `Radius` to the distance value in register 0
- another `LOADWORD` instruction loads 1000, converts it to floating point, and stores it in register 0
- the `FDIV0` instruction divides `Radius` by value 1000.0 in register 0

Copy the Code to your Main Program

In this example we are using the BASIC Stamp as the target, so open the 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 V3 IDE to the *tutorial1.bs2* file in the BASIC Stamp Editor.

Since we don't actually have the sensor described, we'll 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 V3 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
```

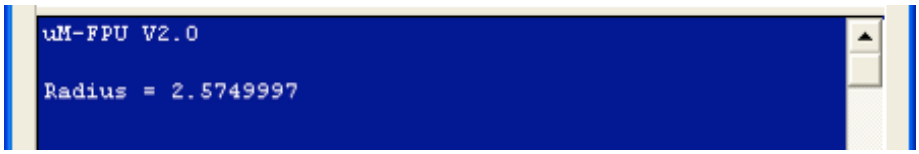
Your program should look like the following (not including definitions and support routines from the template file):

```
'=============================================================================
'==================== 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, [SELECTA, Radius,
    LOADWORD, distance.HIGHBYTE, distance.LOWBYTE, FSET0, LOADWORD, $03, $E8,
    FDIV0]

  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.



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 π (pi) times the diameter. The IDE has a pre-defined name for π, called PI, so you can simple enter the following line to calculate circumference:

```
      Circumference = PI * Diameter
```

The area of a circle is equal to π times radius^2. 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 first add the following definition for the microcontroller variable:

```
areaIn          VAR  Word
```

Next, we add the following line to 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 Main Program

Compile the new code, and copy the generated code from the IDE to the BASIC Stamp program. Add additional DEBUG statements (as described above) to print the new results. Your BASIC Stamp program should now look like the following (not including definitions and support routines 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, [SELECTA, Radius,
    LOADWORD, distance.HIGHBYTE, distance.LOWBYTE, FSET0, LOADWORD, $03, $E8,
    FDIV0]
  DEBUG CR, "Radius:         "
  format = 63
  GOSUB Print_FloatFormat

  ' Diameter = Radius * 2
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, Diameter, FSET, Radius, FMULI, 2]
  DEBUG CR, "Diameter:       "
  format = 63
  GOSUB Print_FloatFormat

  ' Circumference = PI * Diameter
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, Circumference, LOADPI, FSET0,
    FMUL, Diameter]
  DEBUG CR, "Circumference:  "
  format = 63
  GOSUB Print_FloatFormat

  ' Area = PI * Radius * Radius
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, Area, LOADPI, FSET0, FMUL, Radius,
    FMUL, Radius]
  DEBUG CR, "Area:           "
  format = 63
  GOSUB Print_FloatFormat

  '--- areaIn = Area
  ' areaIn = Area
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, 0, LOAD, Area, FIX]
  GOSUB Fpu_Wait
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [LREADWORD]
  SHIFTIN FpuIn, FpuClk, MSBPRE, [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:

`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 source file, 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:

Save the Source File

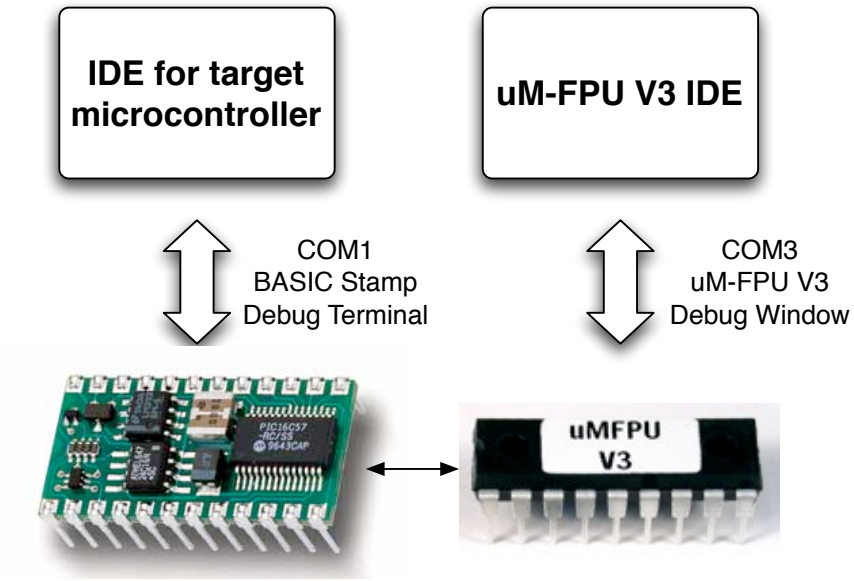
Select the Save command from the File menu, enter the name *tutorial1.fpu* in the save dialog, and click the Save button.

This completes the tutorial on generating uM-FPU V3 code. With the information gained from this tutorial, and more detailed information from the reference section, you should now be able to use the IDE to create your own programs.

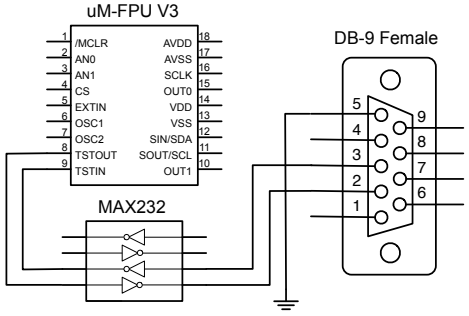
Tutorial 2: Debugging uM-FPU V3 Code

The figure below shows the process of debugging uM-FPU V3 code using the IDE. The left side of the diagram shows the normal BASIC Stamp debug connection using a serial port (e.g. COM1) with the terminal window in the BASIC Stamp Editor. The right side of the diagram shows the additional connection (e.g. COM3) used to connect the uM-FPU V3 IDE to the built-in debug monitor on the uM-FPU V3 chip.

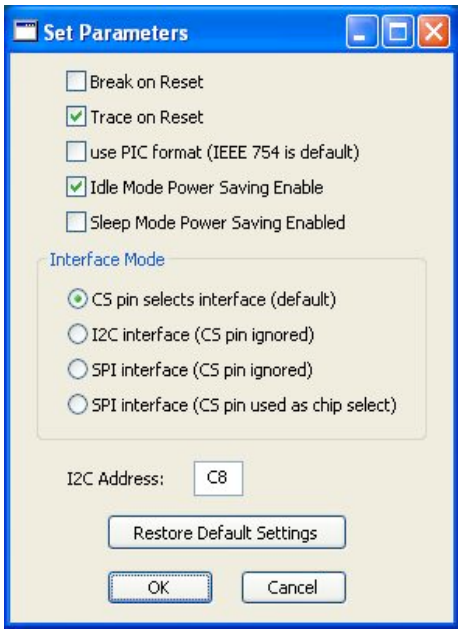
Debugging uM-FPU V3 code



To use the debugger, the uM-FPU V3 IDE requires a 57,600 baud serial connection to the uM-FPU V3 chip configured as 8 data bits, no parity, and 1 stop bit. An example of the connection is shown below.

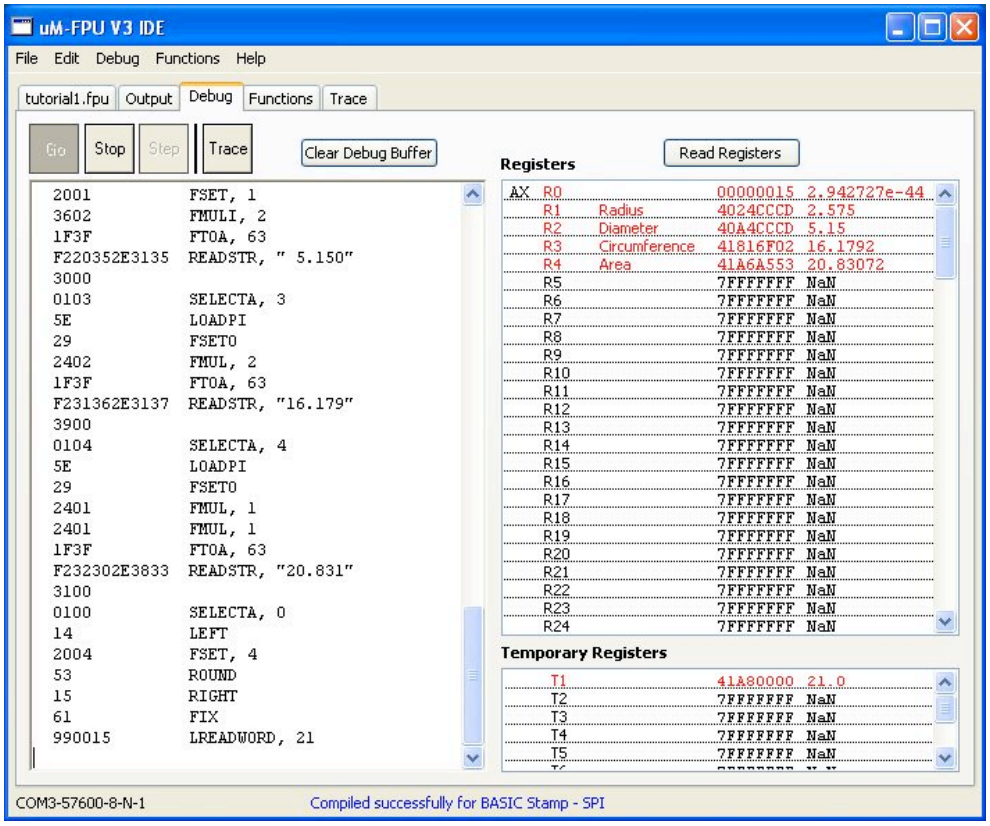


The Trace on Reset parameter determines whether debug tracing is enabled at Reset. Since tracing is disabled by default, we will select the Set Parameters… item from the Functions menu, and make sure that the Trace on Reset option is selected. Check that the Set Parameters dialog looks the same as the 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 V3 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. Click the Read Registers button. The trace buffer and register panel 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. Tracing is very useful for checking the actual sequence of instruction executed by the uM-FPU V3 chip. 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, [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 

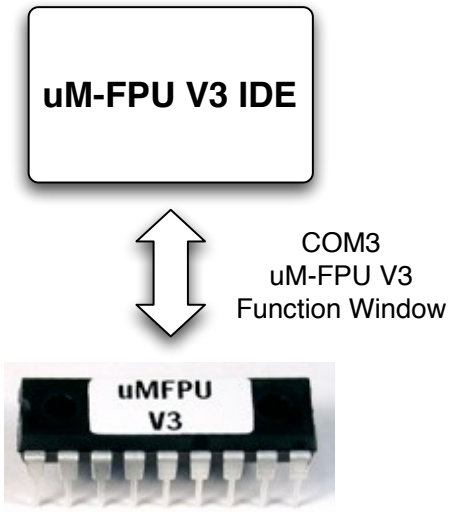
button to continue execution. Check the section entitled *Reference Guide: Debugging uM-FPU V3 Code* for more information.

This completes the tutorial on debugging uM-FPU V3 code. With 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.

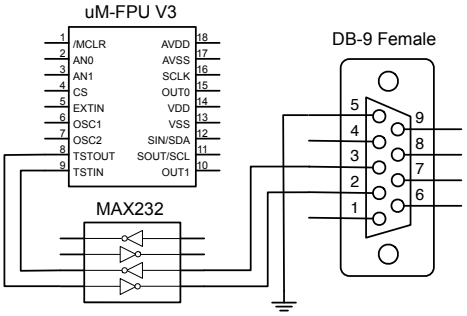
Tutorial 3: Storing User-Defined Functions

The figure below shows the process of storing user-defined functions. The IDE provides support for reading, displaying a memory map, and programming the Flash memory used to store user-defined functions on the uM-FPU V3 chip .

Storing User-Defined Functions.



To store user-defined functions, the IDE requires a 57,600 baud serial connection to the uM-FPU V3 chip configured as 8 data bits, no parity, and 1 stop bit. An example of the connection is shown below.



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 V3 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. 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
```

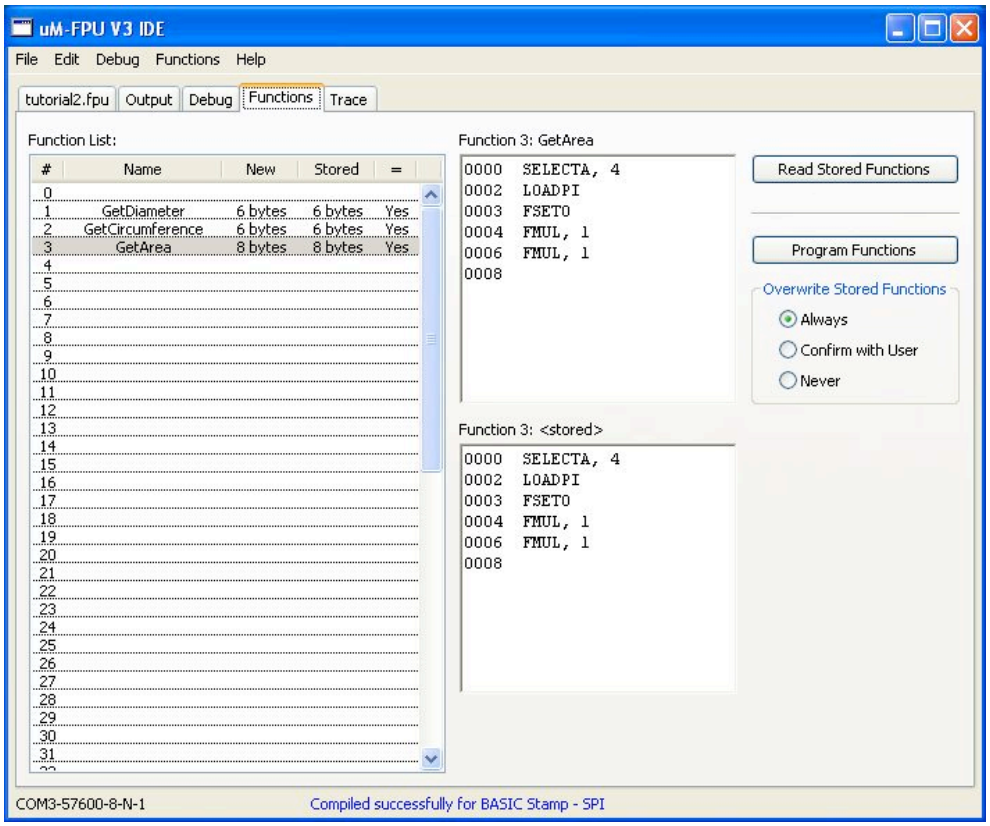
Modifying the Code for Functions

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



The Functions window above shows that three functions have been defined. Code generated for the selected function is displayed in the upper middle panel. If that function has been previously stored on the uM-FPU V3 chip, the stored code will be displayed in the lower middle panel. If a function has not been previously stored on the uM-FPU V3 chip, the lower middle panel will be empty. Select 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 Program Functions button to store the user-defined functions on the uM-FPU V3 chip. A status dialog will be displayed as the user-defined functions are programmed into Flash memory on the uM-FPU V3 chip. If an error occurs, check the connection, turn the power to the uM-FPU V3 chip on and off to ensure that it has been reset, and try again.

Running the Program

Copy the generated code from the uM-FPU V3 IDE to the BASIC Stamp program, replacing the diameter, circumference and area calculations with function calls. Remember to also copy the uM-FPU Function definitions. The main routine in your BASIC Stamp program should now look as follows:

```
'-------------------- 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

'-------------------- uM-FPU Function Definitions -----------------------------
GetDiameter     CON     1               ' uM-FPU user function 1
GetCircumference CON    2               ' uM-FPU user function 2
GetArea         CON     3               ' uM-FPU user function 3

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

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

Main:
  distance = 2575

  ' Radius = distance / 1000
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, Radius,
    LOADWORD, distance.HIGHBYTE, distance.LOWBYTE, FSET0, LOADWORD, $03, $E8,
    FDIV0]
  DEBUG CR, "Radius:         "
  format = 63
  GOSUB Print_FloatFormat

  ' @GetDiameter
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FCALL, GetDiameter]
  format = 63
  GOSUB Print_FloatFormat

  ' @GetCircumference
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FCALL, GetCircumference]
  DEBUG CR, "Circumference:  "
  format = 63
  GOSUB Print_FloatFormat

  ' @GetArea
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FCALL, GetArea]
  DEBUG CR, "Area:           "
  format = 63
  GOSUB Print_FloatFormat

  ' areaIn = ROUND(area)
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, 0, LEFT, FSET, Area, ROUND, RIGHT,
    FIX]
  GOSUB Fpu_Wait
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [LREADWORD]
  SHIFTIN FpuIn, FpuClk, MSBPRE, [areaIn.HIGHBYTE, areaIn.LOWBYTE]
  DEBUG CR, "AreaIn:         ", DEC AreaIn

  END
```

Save the IDE 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:

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

Since calling an undefined functions has no effect, register A remains unchanged after the `Radius` calculation, and the same value prints out for each `Print_Format` call. The `AreaIn` value is displayed as 65535 because the value of `Area` is `NaN`, so `AreaIn` is returned as `-1`.

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

Reference Guide: Generating uM-FPU V3 Code

The source file is created to enter the definitions and math expressions for 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. e.g.

```
F1 = F2 + F3 * F4
```

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

Parentheses are used to change the order of operations. e.g.

```
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 in the expression. e.g.

```
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 is incorrect.

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

Pre-defined Register Names

The following register names are pre-defined:

`F0, F1, F2, … F127`	specifies that register 0 to 127 contains a floating point value
`L0, L1, L2, … L127`	specifies that register 0 to 127 contains a long integer value
`U0, U1, U2, … U127`	specifies that register 0 to 127 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, ASIN, ACOS, ATAN, ATAN2,
DEGREES, RADIANS, FLOOR, CEIL, ROUND, POWER, ROOT, FRAC, INV, FLOAT, FIX,
FIXR, ABS, MOD, MIN, MAX
```

User-Defined Names

Names can make expressions much easier to read and understand. 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 microcontroller variables are used in expressions, the IDE generates the necessary code to transfer the value between the microcontroller and the uM-FPU. For example, the following input would generate code to load `degreesC` from the microcontroller, convert it to floating point, multiply it by 1.8 and then add 32.

e.g.

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

Note: When writing code for the PICAXE, variable definitions must include the PICAXE register used for the variable.

e.g.

```
degreesC     EQU    BYTE   b3
degreesF     EQU    UWORD  w0
```

Comments

Comments are prefaced by an apostrophe (`'`) , double slash (`//`) or semicolon (`;`). The IDE will ignore all characters from the comment prefix to the end of the current input line.

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, or by the name of a previously defined function.
e.g.

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

With the uM-FPU V3 chip, when a function is called from inside another function, execution returns to the original function when the called function finishes. Function calls can be nested up to 16 levels deep.

Function Prototypes

An alternate way to define functions is to define function prototypes. By placing prototypes at the top of the source code, functions can be defined and called in any order, since the function values are known. Function prototypes are defined using the FUNC operator followed by a function number. e.g.

```
Hypotenuse FUNC 1
...

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

The % character can be also be used instead of a number to automatically allocate the next available function.

Entering uM-FPU Assembler Code

The IDE normally regular math expressions and compiles them to produce the required uM-FPU V3 instructions. Some capabilities of the uM-FPU V3 chip are not accessible using math expressions, or in some cases it may be desirable to write more optimized code using assembler. Assembler code can be entered by enclosing it with the `#ASM` and `#ENDASM` directives. The syntax of 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. For example:

```
#ASM SELECTA, 1 LOADPI FSET #ENDASM
```

or

```
#ASM
  SELECTA, 1
  LOADPI
  FSET
#ENDASM
```

Assembler Instructions

```
NOP
SELECTA, reg
SELECTX, reg
CLR, reg
CLRA
CLRX
CLR0
COPY, reg, reg
COPYA, reg
COPYX, reg
LOAD, reg
LOADA
LOADX
ALOADX
XSAVE, reg
XSAVEA
COPY0, reg
COPYI, bb, reg
SWAP, reg, reg
SWAPA, reg
LEFT
RIGHT
FWRITE, reg, floatval
FWRITEA, floatval
FWRITEX, floatval
FWRITE0, floatval
FREAD
FREADA
FREADX
FREAD0
ATOF, string
FTOA, bb
FSET, reg
FADD, reg
FSUB, reg
FSUBR, reg
FMUL, reg
FDIV, reg
FDIVR, reg
FPOW, reg
FCMP, reg
FSET0
FADD0
FSUB0
FSUBR0
FMUL0
FDIV0
FDIVR0
FPOW0
FCMP0
FSETI, bb
FADDI, bb
FSUBI, bb
FSUBRI, bb
FMULI, bb
FDIVI, bb
FDIVRI, bb
FPOWI, bb
FCMPI, bb
FSTATUS, reg
FSTATUSA
FCMP2, reg, reg
FNEG
FABS
FINV
SQRT
ROOT, reg
LOG
LOG10
EXP
EXP10
SIN
COS
TAN
ASIN
ACOS
ATAN
ATAN2, reg
DEGREES
RADIANS
FMOD, reg
FLOOR
CEIL
ROUND
FMIN, reg
FMAX, reg
FCNV, bb
FMAC, reg, reg
FMSC, reg, reg
LOADBYTE bb
LOADUBYTE bb
LOADWORD wwww
LOADUWORD wwww
LOADE
LOADPI
LOADCON, bval
FLOAT
FIX
FIXR
FRAC
FSPLIT
SELECTMA, reg, bb, bb
SELECTMB, reg, bb, bb
SELECTMC, reg, bb, bb
LOADMA, bb, bb
LOADMB, bb, bb
LOADMC, bb, bb
MOP, bb
LOADIND, reg
SAVEIND, reg
INDA, reg
INDX, reg
FCALL, fnum
EECALL, fnum
```

```
RET
BRA, bb
BRA, cc, bb
JMP, wwww
JMP, cc, wwww
TABLE, bb
FTABLE, bb
LTABLE, bb
POLY, bb
GOTO, reg
LWRITE, reg, longval
LWRITEA, longval
LWRITEX, longval
LWRITE0, longval
LREAD
LREADA
LREADX
LREAD0
LREADBYTE
LREADWORD
ATOL, string
LTOA, bb
LSET, reg
LADD, reg
LSUB, reg
LMUL, reg
LDIV, reg
LCMP, reg
LUDIV, reg
LUCMP, reg
LTST, reg
LSET0
LADD0
LSUB0
LMUL0
LDIV0
LCMP0
LUDIV0
LUCMP0
LTST0
LSETI, bb
LADDI, bb
LSUBI, bb
LMULI, bb
LDIVI, bb
LCMPI, bb
LUDIVI, bb
LUCMPI, bb
LTSTI, bb
LSTATUS, reg
LSTATUSA
LCMP2, reg, reg
LUCMP2, reg, reg
LNEG
LABS
LINC, reg
LDEC, reg
LNOT
LAND, reg
LOR, reg
LXOR, reg
LSHIFT, reg
LMIN, reg
LMAX, reg
LONGBYTE bb
LONGUBYTE bb
LONGWORD wwww
LONGUWORD wwww
LONGCON, bb
SETOUT, bb
ADCMODE, bb
ADCTRIG
ADCSCALE, bb
ADCLONG, bb
ADCLOAD, bb
ADCWAIT
TIMESET
TIMELONG
TICKLONG
EESAVE, reg, bb
EESAVEA, bb
ELOAD, reg, bb
ELOADA, bb
EEWRITE, bb
EXTSET
EXTLONG
EXTWAIT
STRSET, string
STRSEL, bb, bb
STRINS, string
STRCMP, string
STRFIND, string
STRFCHR, string
STRFIELD, bb
STRTOF
STRTOL
READSEL
SYNC
READSTATUS
READSTR
VERSION
IEEEMODE
PICMODE
CHECKSUM
BREAK
TRACEOFF
TRACEON
TRACESTR, string
TRACEREG, reg
READVAR, bb
RESET
```

Where:

`reg`	register number (0-127)
`fnum`	Flash function number (0-63)
`bb`	8-bit value
`wwww`	16-bit value
`cc`	condition code `(Z,EQ,NZ,NE,LT,LE,GT,GE,PZ,MZ,INF,FIN,PINF,MINF,NAN,TRUE,FALSE)`
`floatval`	floating point value
`longval`	long integer value
`string`	ASCII string

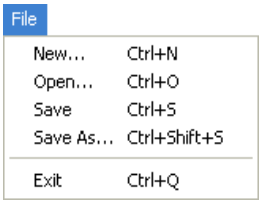
Assembler Directives

`#BYTE bb`	8-bit byte value
`#WORD wwww`	16-bit word value
`#LONG longval`	long integer value
`#FLOAT floatval`	floating point value

Wait Code

The uM-FPU V3 chip has a 256 byte instruction buffer. If the instructions and data in a calculation exceed 256 bytes, the buffer could overflow, so the program must wait for the buffer to empty at least every 256 bytes. The code generated by the IDE accounts for this, and will insert a wait sequence as required. Read operations automatically generate a wait sequence, so in many applications, no additional wait sequences are required.

File Menu



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, you 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, you 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.

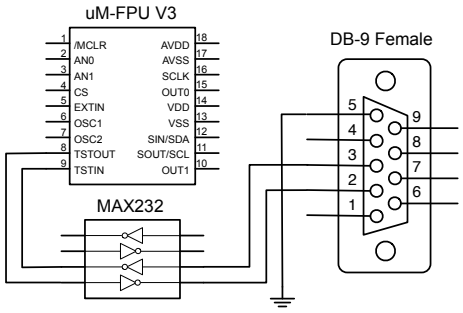
The Exit menu item causes the IDE to quit. If a source file is open, and has been changed since the last time it was saved, you will first be prompted to save the source file.

Reference Guide: Debugging uM-FPU V3 Code

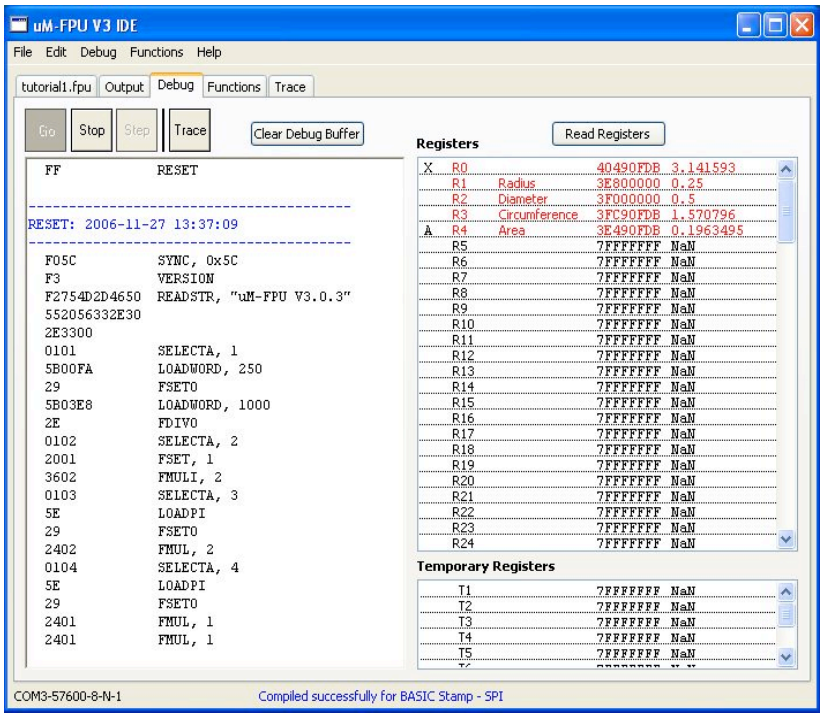
Utilizing the built-in debug monitor on the uM-FPU V3 chip, the IDE provides a high-level interface for debugging programs that use the uM-FPU V3 floating point coprocessor. It supports the ability to trace uM-FPU instructions, set breakpoints, single-step through execution of uM-FPU instructions, and 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 IDE must be connected to the uM-FPU V3 chip using a 57,600 baud serial connection with 8 data bits, no parity, and 1 stop bit. An example of the connection is shown below.



Debug Window



The scrolling window on the left displays trace messages, and the panel on the right displays the contents of the uM-FPU registers. The 

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 debug trace output. When a Reset occurs a message is displayed showing the date and time of the Reset.
e.g.

```
----------------------------------------
RESET: 2006-11-28 13:19:31
----------------------------------------
```

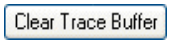
Tracing is turned off at Reset, unless the Trace on Reset parameter has been set. Tracing can be controlled by the program using the `TRACEON` and `TRACEOFF` instructions, or manually with the 

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
  0104           SELECTA, 4
  5E             LOADPI
  29             FSET0
  2401           FMUL, 1
  2401           FMUL, 1
  1F3F           FTOA, 63
  F232302E3833   READSTR: "20.831"
  3100
  …
```

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 manually with the 

button. Breakpoints occur after the next uM-FPU instruction finishes executing. When a breakpoint occurs, the last uM-FPU instruction executed before the breakpoint is displayed, followed by the break message, and the register display is updated. 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.
e.g.

```
  5E             LOADPI
BREAK
```

The 

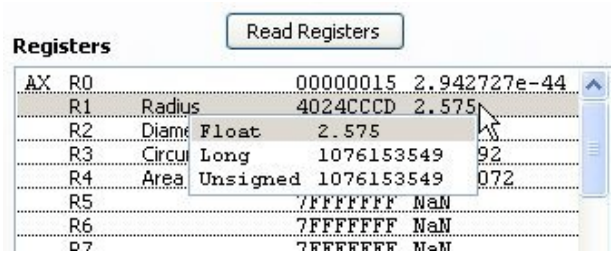
buttons are enabled or disabled depending on the current state of execution. The Go button is used to continue execution, and is enabled at Reset or after a breakpoint occurs. The Stop button is used to stop execution after the next uM-FPU instruction is executed. If the uM-FPU is idle when the Stop 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, with a new breakpoint occurring after each instruction.

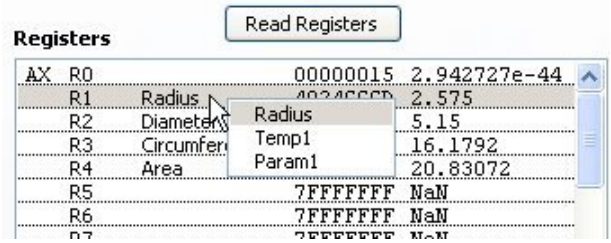
The Register Panel

The register panel displays the value of each register and indicates the register currently selected as register A and register X. Register A and register X are indicated by an `A` and `X` marker in the left margin of the register panel. The temporary registers are displayed at the bottom on the register panel.

For each register, the register number, optional register name, hexadecimal value, and formatted value is displayed. If you click on the formatted value, a pop-up menu is displayed with the register value displayed in floating point, long integer, and unsigned long integer format. If you select a different format, the display will be updated to show that format.
e.g.



Register names are automatically set from the register definitions in the source file. Registers can often have several different names assigned. If you click on the register name, a pop-up menu is displayed showing all of the names for that register. If you select a different name, the display will be updated to show that name.
e.g.

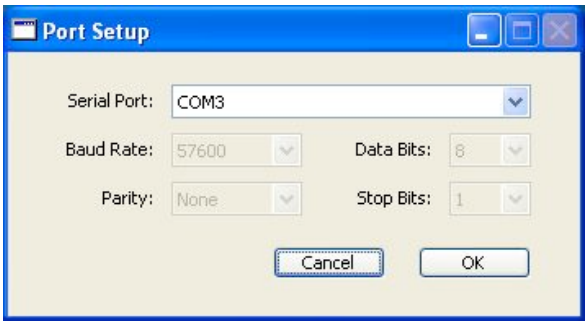


The current register values are automatically updated after every breakpoint. The Read Registers button can also be used to manually force an update of the register values. Register values are displayed in red if the value has changed since the last time the display was updated, or black if the value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

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

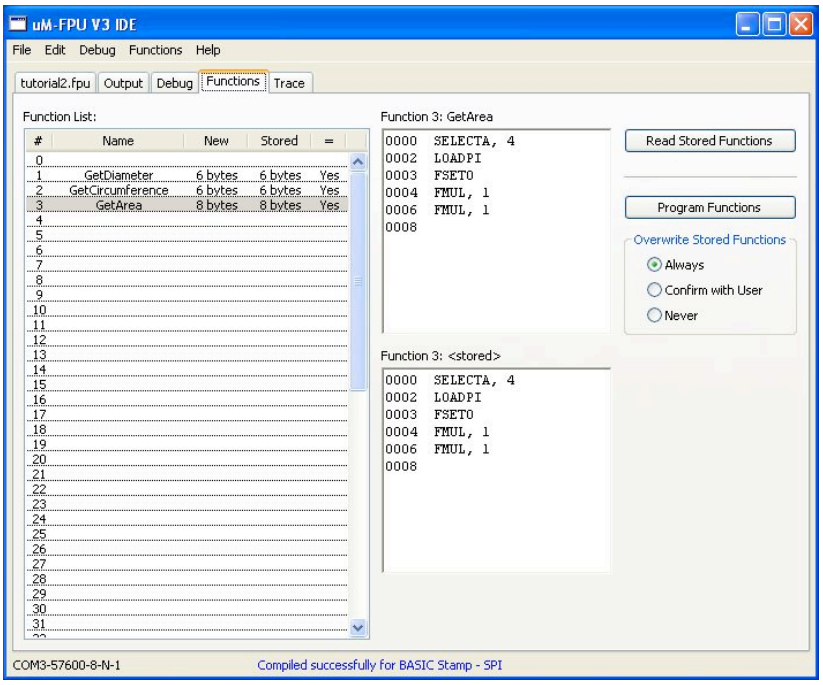
The Read Version menu item will display the version of the uM-FPU in the trace window.

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

Reference Guide: Storing User-Defined Functions

The Functions window provides support for storing user-defined functions on the uM-FPU V3 chip. Stored functions can reduce memory usage on the microcontroller, simplify the interface and often increase the speed of operation. The uM-FPU V3 reserves 2048 bytes of flash memory for user-defined functions and parameters (plus 256 bytes for the header information). 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. See the section entitled *Reference Guide: Generating uM-FPU V3 Code* for more details.

Function Window

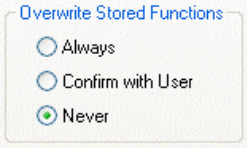


The scrolling list on the left shows all of the currently defined functions. The Name column displays the name of any function names defined in the source file. The New column shows the size in bytes of the functions defined in the source file, and the Stored column displays the size in bytes of functions currently stored on the uM-FPU V3 chip (if the functions have been read). 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 functions defined in the source file, and the panel located at bottom center displays the uM-FPU instructions for the functions stored on the uM-FPU V3 chip. The function to be displayed is selected by clicking on one of the functions in the Function List.

Clicking the Read Stored Functions button reads all of the functions currently stored on the uM-FPU V3 chip and updates the function list.

Clicking the Program Functions button programs all of functions defined in the function list into Flash memory on the uM-FPU V3 chip. If a newly defined function is different then 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.

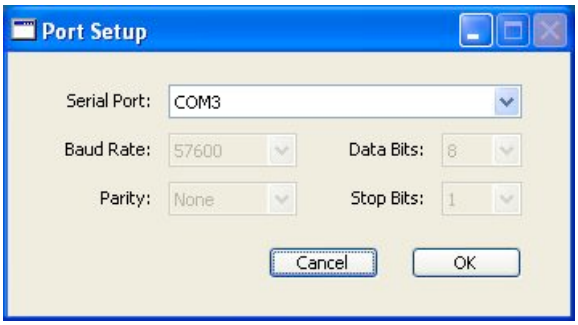
If the Confirm with User option is selected, you are asked to confirm whether a new function should replace the previously stored function.

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

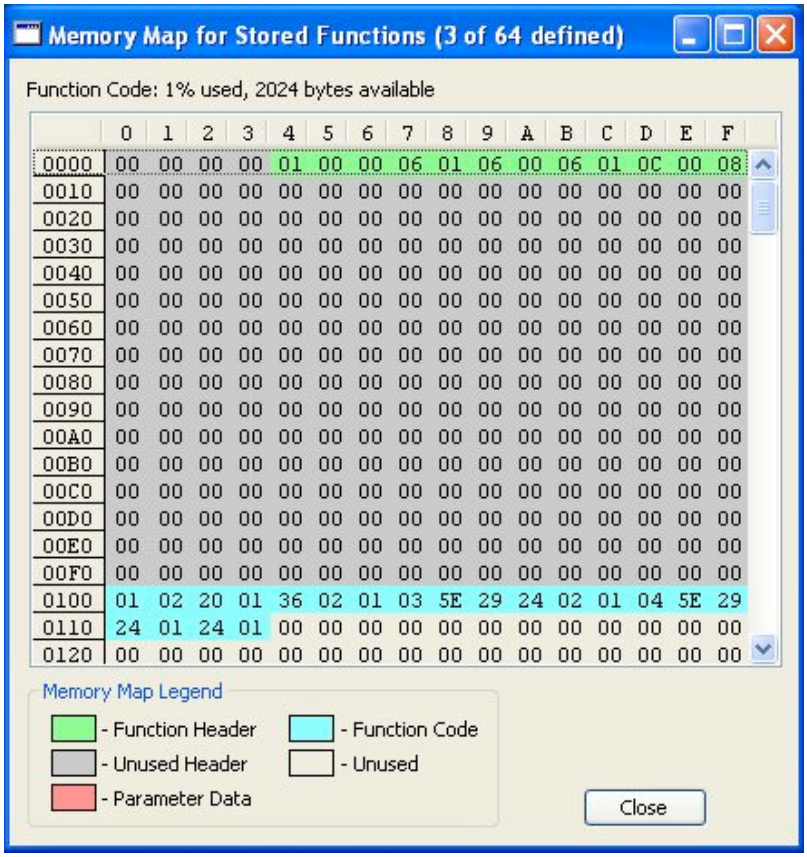


The Read Stored Functions menu item has the same function as the Read Stored Functions button.

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 Flash memory reserved for user-defined functions on the uM-FPU V3 chip. A status line at the top shows the percent of memory used and the number of bytes available.

Memory Map Dialog



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	00	00	00	00	01	00	00	06	01	06	00	06	01	0C	00	08
0010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0100	01	02	20	01	36	02	01	03	5E	29	24	02	01	04	5E	29
0110	24	01	24	01	00	00	00	00	00	00	00	00	00	00	00	00
0120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

The Clear Functions menu item will clear all of the user-defined functions from Flash memory on the uM-FPU V3 chip. A dialog will be displayed requesting confirmation before the functions are cleared from memory.

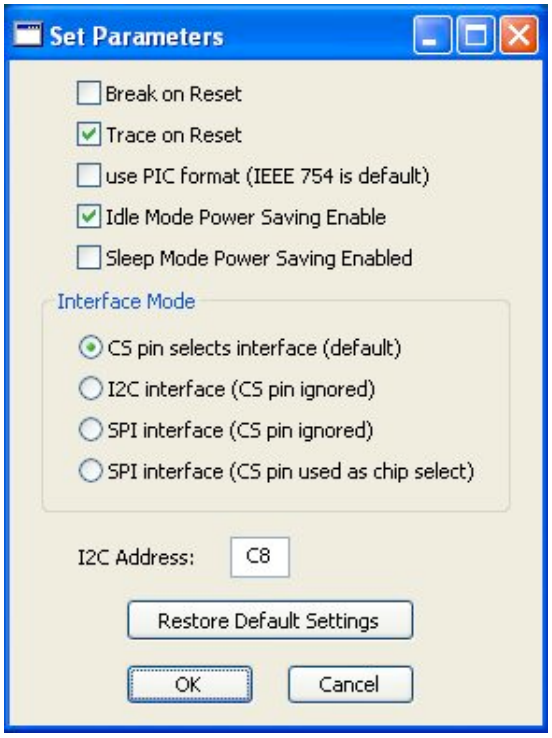
The Program Functions menu item has the same function as the Program Functions button.

The Set Parameters… menu item is used to set the uM-FPU V3 parameters as described in the next section.

Reference Guide: Setting uM-FPU V3 Parameters

The Set Parameters… menu item is used to set the uM-FPU V3 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. The uM-FPU V3 chip 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. An 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 V3 chip or written to the uM-FPU V3 chip, and the microcontroller program can use the PIC format. The `IEEEMODE` and `PICMODE` instructions can also be used to dynamically change the format. For additional information regarding the `IEEEMODE` and `PICMODE` instructions, see the *uM-FPU V3 Instruction Reference*.

> Note: The IDE code generator currently only generates code for the default IEEE 754 format. If the PIC format is used you will need to fix the data values in the code generated for `FWRITE`, `FWRITEA`, `FWRITEX` and `FWRITE0` instructions.

Idle Mode Power Saving Enable
If this option is selected, the uM-FPU V3 chip will go into a low power mode when idle.

Sleep Mode Power Saving Enabled
If this option is selected, the uM-FPU V3 chip will go to sleep when idle and the chip is not selected. This mode is only active if the interface mode is SPI with the CS pin used as a chip select.

Interface Mode
By default, the CS pin on the uM-FPU V3 chip is read at Reset to determine if the SPI or I^2C interface is to be used. The interface mode parameter can be used to force selection of SPI or I^2C at Reset (ignoring the CS pin), or to specify SPI mode with the CS pin acting as a chip select.

> Note: Most of the SPI support software currently supplied by Micromega assumes that no chip select is used. If the chip select option is enabled, you must ensure that the CS pin is being handled properly. If SPI is used without chip select, the CS pin must be tied low.

I2C Address
By default, the I^2C address used by the uM-FPU V3 chip is `C8` (hexadecimal) or `1100100x` (binary). If the default address conflicts with another I^2C device, or if multiple uM-FPU V3 chips are used on the same I^2C bus, the address can be changed to any other valid I^2C 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
Idle Mode Power Saving
No Sleep Mode Power Saving
CS pin selects I^2C or SPI
I^2C address is `C8`

Further Information
The following documents are also available:

uM-FPU V3 Datasheet	provides hardware details and specifications
uM-FPU V3 Instruction Reference	provides detailed descriptions of each instruction

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