Micromega Corporation

uM-FPU64 IDE
Integrated Development Environment

User Manual
Release 404

Introduction

The uM-FPU64 Integrated Development Environment (IDE) software provides a set of easy-to-use tools for developing applications using the uM-FPU64 floating point coprocessor. The IDE runs on Windows XP, Vista and Windows 7, and provides support for compiling, debugging, and programming the uM-FPU64 floating point coprocessor.

Main Features

Compiling

- built-in code editor for entering symbol definitions and math expressions
- compiler generates code customized to the selected microcontroller
- pre-defined code generators included for most microcontrollers
- target description files can be defined by the user for customized code generation
- compiler code and assembler code can be mixed to support all uM-FPU64 instructions
- output code can be copied to the microcontroller program

Debugging

- instruction tracing
- contents of all FPU registers can be displayed in various formats
- breakpoints and single-step execution
- conditional breakpoints using auto-step capability
- symbol definitions from compiler used by instruction trace and register display
- numeric conversion tool for 32-bit and 64-bit floating point and integer values

Programming Flash Memory

- built-in programmer for storing user-defined functions in Flash memory
- memory map display for Flash memory
- graphic interface for setting parameter bytes stored in Flash

Further Information

The following documents are also available:

uM-FPU64 Datasheet	provides hardware details and specifications
uM-FPU64 Instruction Set	provides detailed descriptions of each instruction

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

Table of Contents

Installing the uM-FPU64 IDE Software

The uM-FPU64 IDE software can be downloaded from the Micromega website at:

http://www.micromegacorp.com/umfpu64-ide.html

The download is called *uM-FPU64 IDE xxx.zip* (where *xxx* is the release number e.g. *r404*). Double-click or unzip the file, then open the folder, and run the installer called *uM-FPU64 IDE setup.exe*. The software is installed in the *Program FIles>Micromega* folder, and the Start Menu entry is *Micromega*.

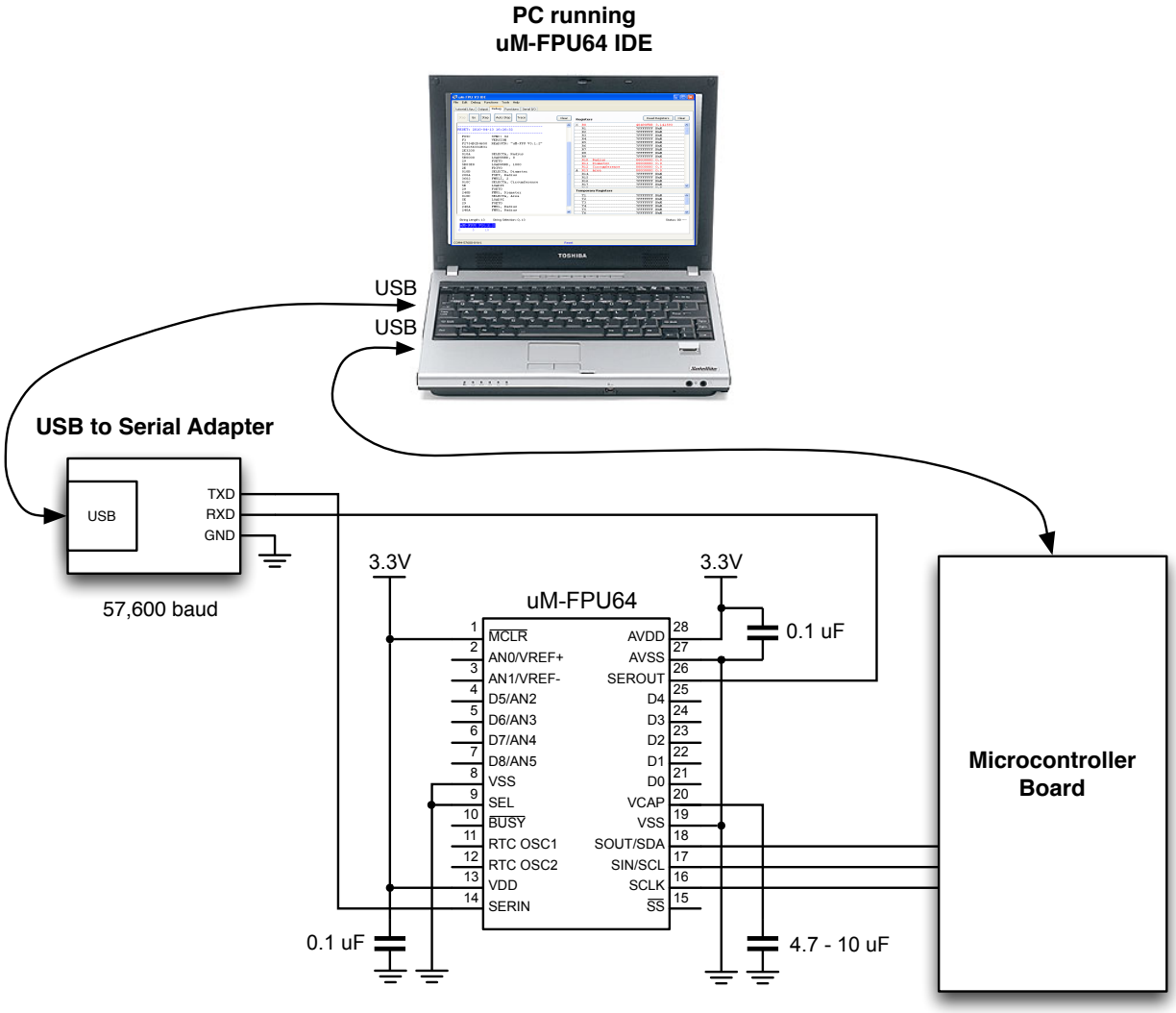
Connecting to the uM-FPU64 chip

Compiling can be done without a serial connection, but a serial connection between the computer running the IDE and the uM-FPU64 chip is required for debugging and programming. For recent computers, the easiest way to add a serial connection is using a USB to Serial adapter. Older computers with serial ports, or USB to RS-232 adapters require a level converter (e.g. MAX232). The uM-FPU64 chip requires a non-inverted serial interface operating at the same voltage as the FPU (i.e. if the FPU is operating at 3.3V, the serial interface must be a 3.3V interface). The IDE communicates with the uM-FPU64 chip at 57,600 baud, using 8 data bits, no parity, one stop bit, and no flow control.

Examples of suitable USB to Serial adapters include:

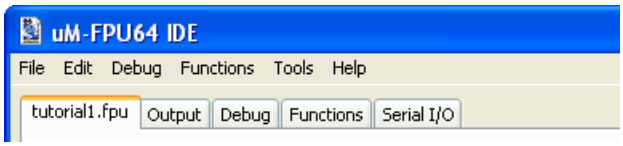
Sparkfun	FTDI Basic Breakout - 3.3V	http://www.sparkfun.com/
Parallax	USB2SER Development Tool	http://www.parallax.com/

Connection Diagram



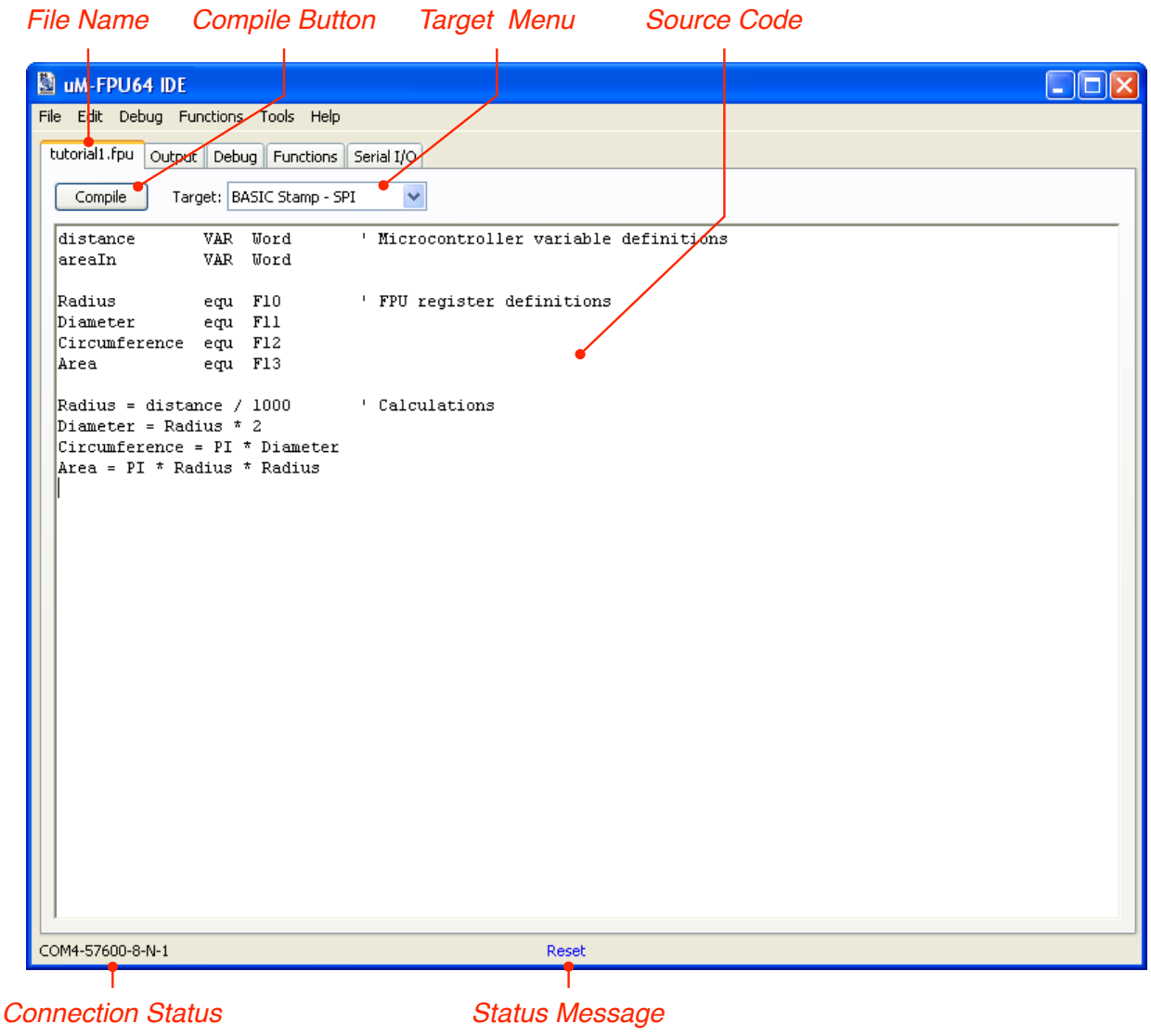
Overview of uM-FPU64 IDE User Interface

The main window of the IDE has a menu bar, and a set of tabs attached to five different windows. Clicking a tab will display the associated window.



Source Window

The ***Source Window*** is the leftmost tab, and the filename 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*. If the source file has been modified since the last save, an asterisk is displayed after the filename. The source file is stored as a text file with a default extension of *fpu*.

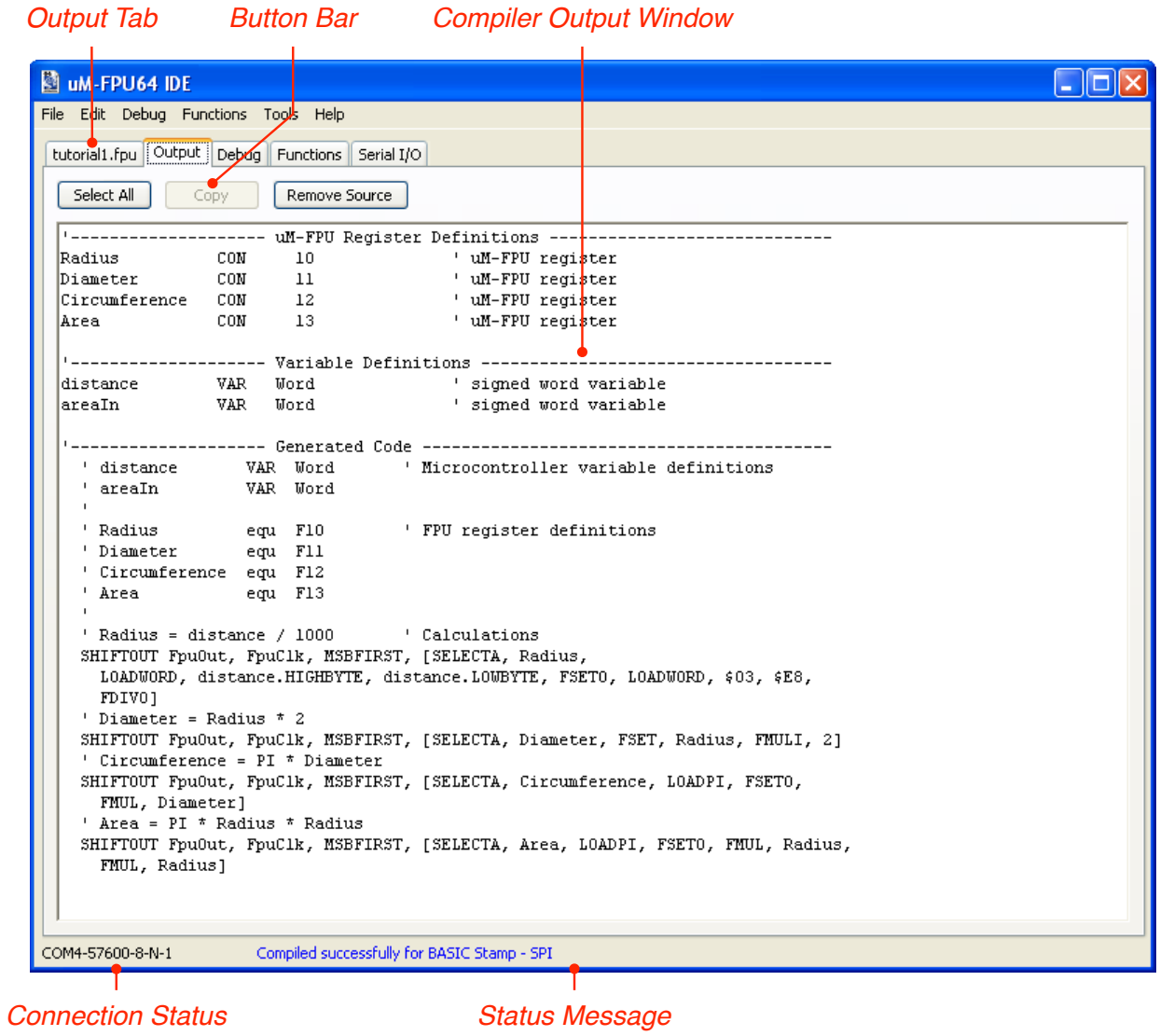


The ***Source Window*** is used to edit the source code and compile the source code. Pressing the ***Compile*** button

will compile the code for the target selected by the ***Target Menu***. If an error occurs during compile, then an error message will be displayed as the ***Status Message***. All error messages are displayed in red.

Output Window

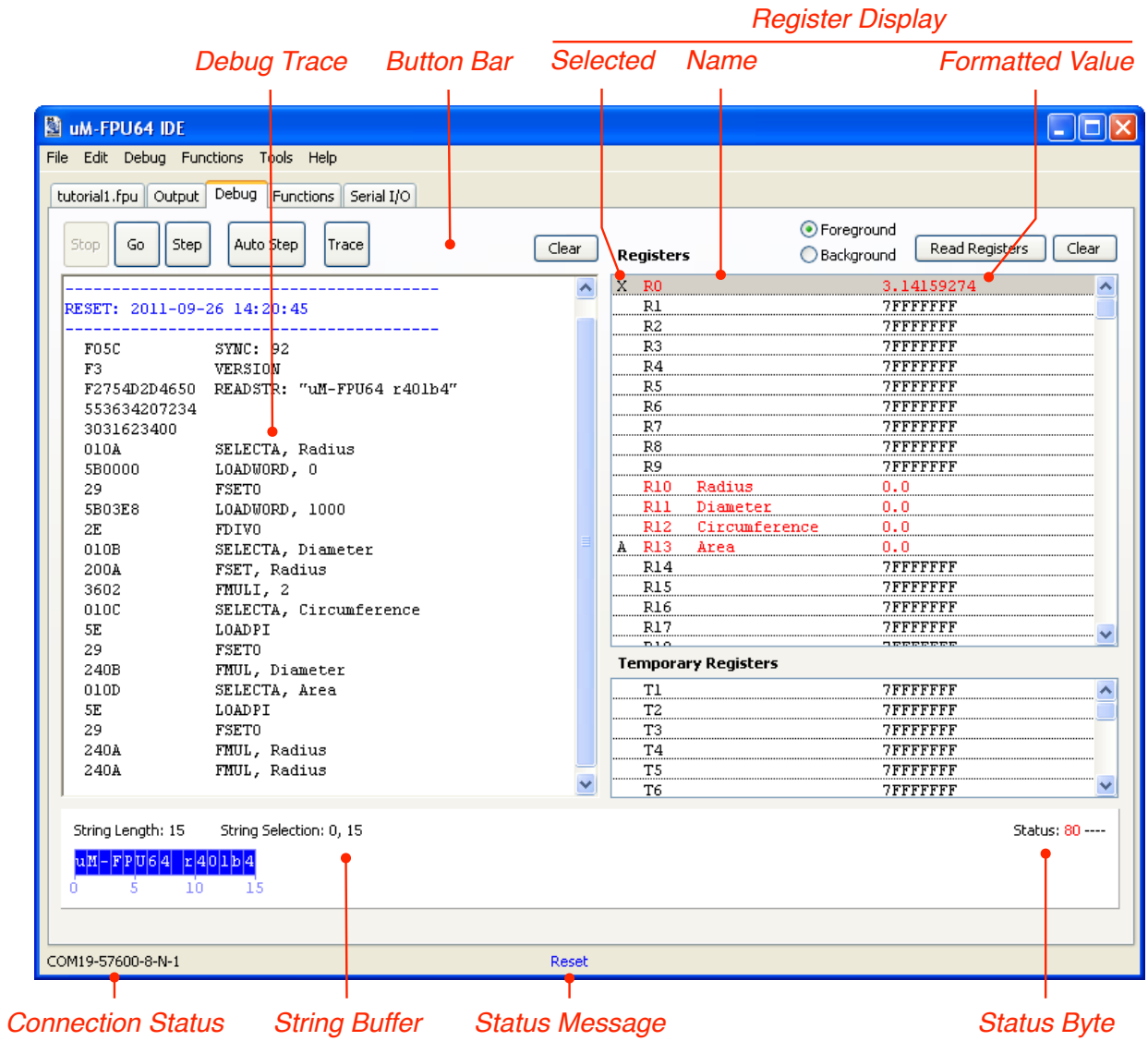
The ***Output Window*** is automatically displayed if the compile is successful. The status message will show that the compile was successful. All normal status messages are displayed in blue.



If the code was generated for a target microcontroller, the ***Select All*** and ***Copy*** buttons can be used to copy the code from the window so it can be pasted into the microcontroller program. Alternatively, the code can be copy-and-pasted a section at a time by doing a text selection and using the ***Copy*** button. The ***Remove Source*** button can be used to remove the source code lines that are included as comments.

Debug Window

The ***Debug Window*** is used for debugging. It displays the instruction trace, reset and breakpoint information, and the contents of the FPU registers, string buffer and status value.

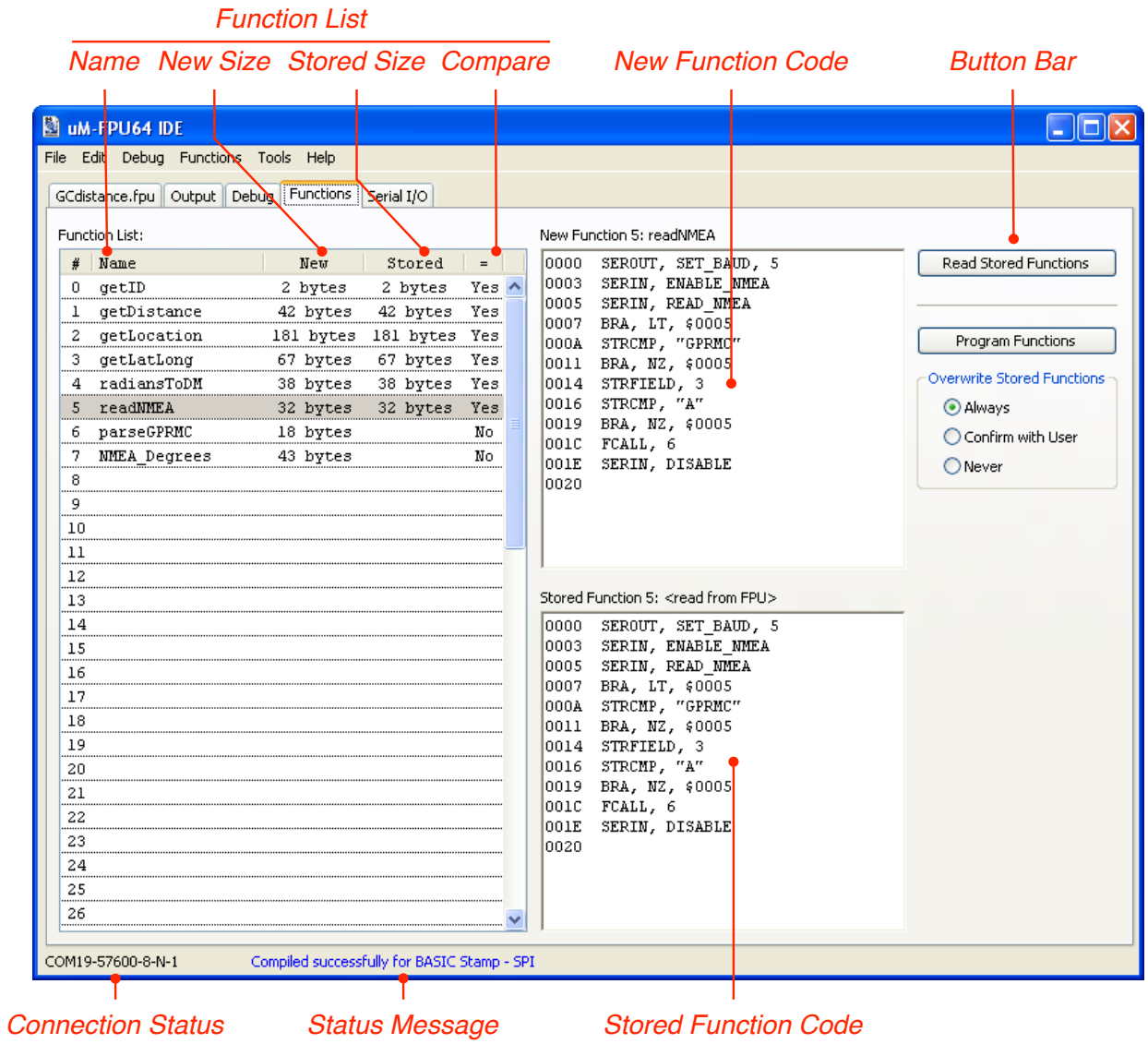


The ***Debug Trace*** displays messages and instruction traces. The Reset message includes a time stamp, is displayed whenever a hardware or software reset occurs. Instruction tracing will only occur if tracing is enabled. This can be enabled at Reset by setting the ***Trace on Reset*** option in the ***Functions>Set Parameters...*** dialog, or at any time by by sending the **`TRACEON`** instruction.

The ***Register Display*** shows the value of all registers. Register values that have changed since the last update are shown in red. The ***String Buffer*** displays the FPU string buffer and string selection, and the ***Status Byte*** shows the FPU status byte and status bit indicators. The ***Register Display***, ***String Buffer***, and ***Status Byte*** are only updated automatically at breakpoints. They can be updated manually using the ***Read Registers*** button.

Functions Window

The ***Functions Window*** shows the function code for all new functions and stored functions. It also can be used to program the functions into Flash memory on the FPU.



The ***Function List*** provides information about each function defined by the compiler and stored on the FPU. The ***New Function Code*** displays the FPU instructions for compiled functions, and the ***Stored Function Code*** displays the FPU instructions for functions stored on the FPU. The ***Read Stored Functions*** button is used to read the functions currently stored on the FPU, and the ***Program Functions*** button is used to program new functions to the uM-FPU64 chip.

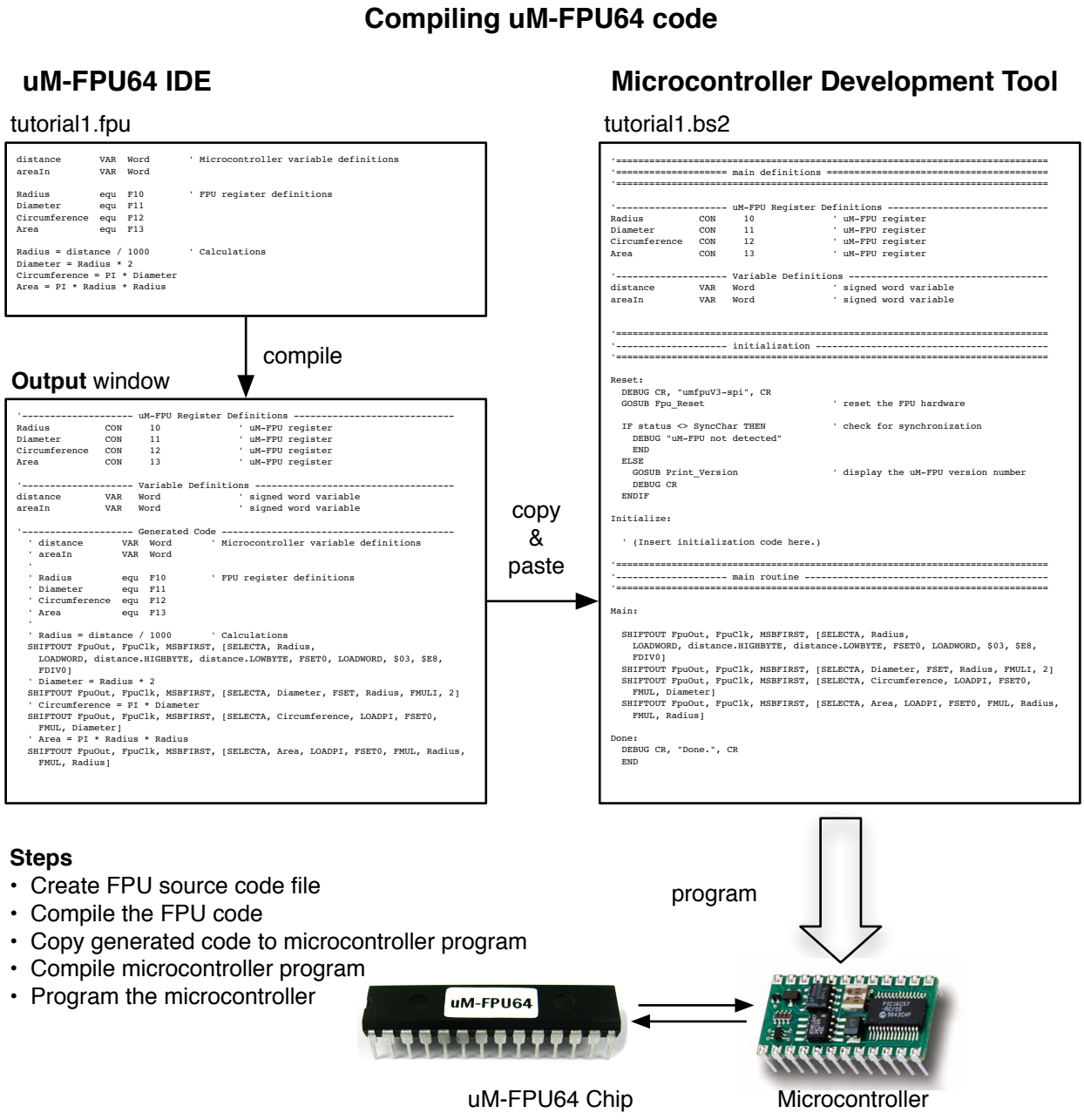
Serial I/O Window

The ***Serial I/O Window*** shows a trace of the serial data exchanged between the IDE and the uM-FPU64 chip. It's provided mainly for diagnostic purposes.

Tutorial 1: Compiling FPU Code

This tutorial takes you through the process of compiling uM-FPU64 code for a few simple examples. Various IDE features are introduced as we go through the tutorial. For a more complete description of specific features, see the the *Reference Guide* sections later in this document.

This tutorial uses the BASIC Stamp with a SPI interface as the target. If you're working with a different microcontroller or compiler, the procedures are the same, but the output code for the selected target will be different. The figure below shows the process of developing FPU code using the IDE.



Starting the uM-FPU64 IDE

Start the uM-FPU64 IDE program. The program will open to an empty ***Source Window*** with the filename set to *untitled*. Since we are using the Basic Stamp for this tutorial, use the ***Target Menu*** to select *BASIC Stamp – SPI*.

The ***Connection Status*** is shown at the lower left of the window**.** A connection is not required to use the compiler, it's only required for debugging and programming.

Entering a Simple Equation

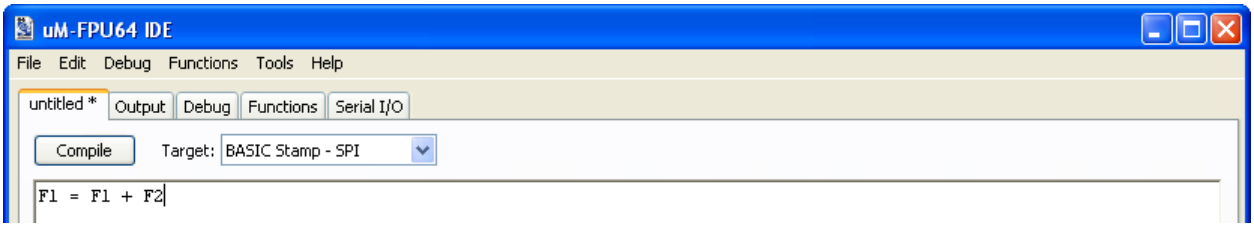
The uM-FPU64 IDE has predefined names for the registers in the FPU.

`F0`, **`F1`**, **`F2`**, … **`F127`** specifies registers 0 through 255, and that the register contains a floating point value
`L0`, **`L1`**, **`L2`**, … **`L127`** specifies registers 0 through 255, and that the register contains a long integer
`U0`, **`U1`**, **`U2`**, … **`U127`** specifies registers 0 through 255, and that the register contains an unsigned long integer

Using these pre-defined names, you can enter a simple equation directly. To add the floating point values in register 1 and register 2, and store the result in register 1, you can enter the following equation:

```
F1 = F1 + F2
```

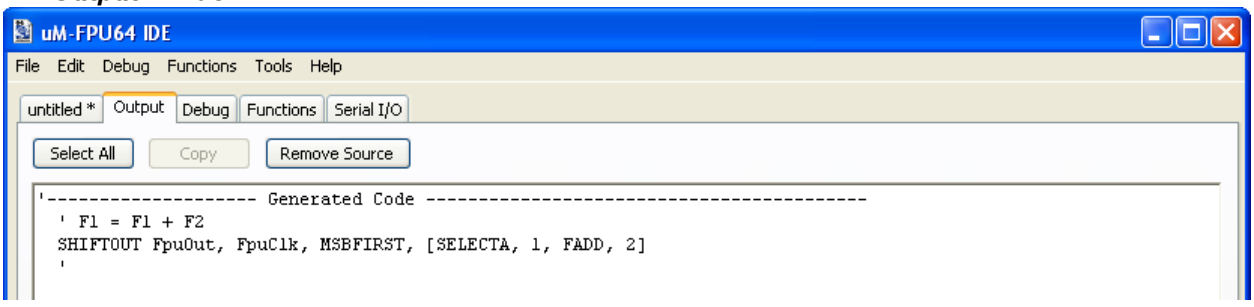
The ***Source Window*** should look as follows:



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 generate up-to-date output code. Click the ***Compile*** button. If the compile is successful, the ***Output Window*** will be displayed, and the status message will be *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 ***Source Window*** above, then click the ***Compile*** button again.

The ***Output Window*** should look as follows:



The expression `F1 = F1 + F2` has been translated into BASIC Stamp code. The code selects FPU register 1 as register A, then adds the value of register 2 to register A. You've successfully compiled your first compile. (If you want to see the code generated for a different target, go back to the ***Source Window*** and select a different target from the ***Target Menu***.)

Defining Names

Math expressions can be easier to read when meaningful names are used. The IDE allows you to define names for 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 FPU. The results are displayed in units of inches to three decimal places.

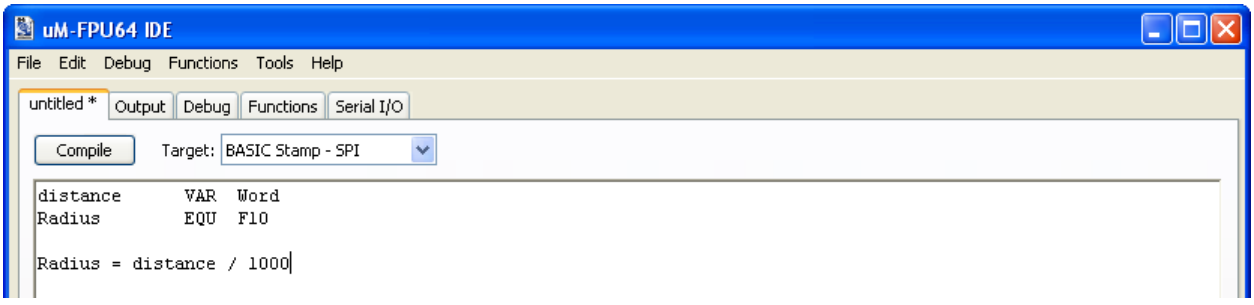
Calculating Radius

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 FPU chip.

Create a new source file using the ***File>New...*** menu item, and enter the following code:

```
distance  VAR  word
Radius    EQU  F10

Radius = distance / 1000
```

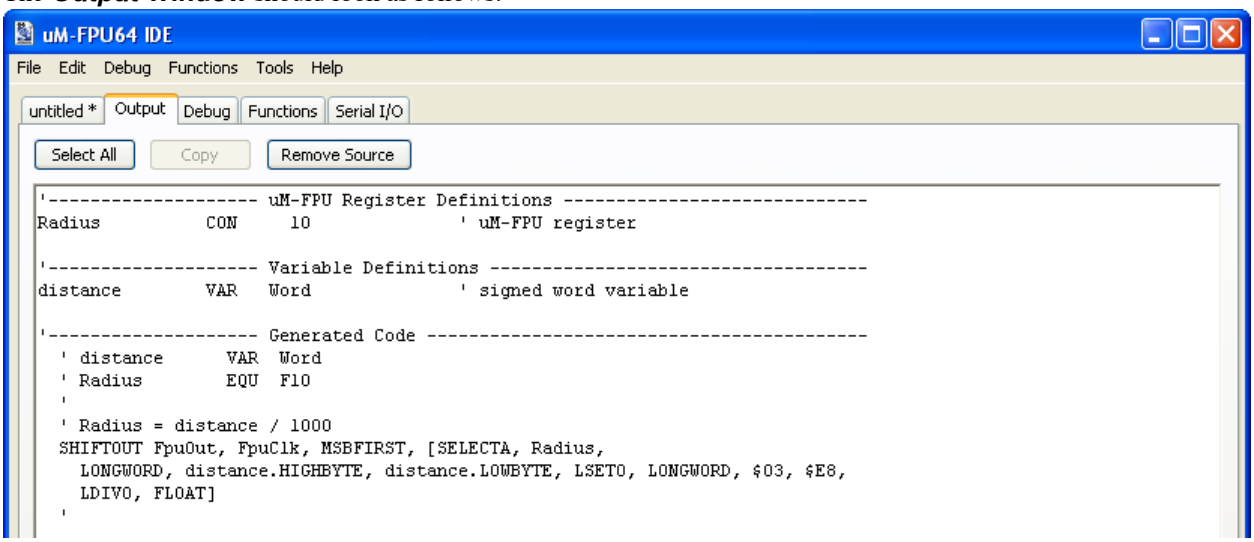
The ***Source window*** should look as follows:



Save the source file using the ***File>Save*** menu item. Save the file as *tutorial1* (with *.fp4* extension added automatically).

Click the ***Compile*** button.

The ***Output Window*** should look as follows:



The generated code does the following:

`SELECTA, Radius`

select the Radius register as register A

`LOADWORD, distance.HIGHBYTE, distance.LOWBYTE, FSET0`

load the 16-bit `distance` *variable to the FPU, convert it to floating point, and store in Radius register*

`LOADWORD, $03, $E8, FDIV0`

load the constant 1000 (hexadecimal value $03, $E8), convert it to floating point, and divide the Radius register by that value

Copying Code to your Main Program

In this example we are using the BASIC Stamp as the target, so open the BASIC Stamp Editor and open the template file *umfpu-spi.bs2*. Save a new copy called *tutorial1.bs2*.

Copy the *uM-FPU Register Definitions* and *Variable Definitions* from the ***Output Window*** and paste in the Basic Stamp program in the *main definitions* section.

Copy the Generated Code from the ***Output Window*** and paste in the Basic Stamp program after the *Main* label.

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 *Main* label.

```
distance = 2575
```

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

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

The main section of your BASIC Stamp program should look as follows:

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

'-------------------- uM-FPU Register Definitions ----------------------------
Radius          CON     10              ' uM-FPU register

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


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

Reset:
  DEBUG CR, "umfpu64-spi", CR
  GOSUB Fpu_Reset                       ' reset the FPU hardware

  IF status <> SyncChar THEN            ' check for synchronization
    DEBUG "uM-FPU not detected"
    END
  ELSE
    GOSUB Print_Version                 ' display the uM-FPU version number
    DEBUG CR
  ENDIF

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

Main:
  distance = 2575

'-------------------- Generated Code -----------------------------------------
  ' distance       VAR  Word
  ' Radius         equ  F10
  '
  ' 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

Done:
  DEBUG CR, "Done.", CR
  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 in the start of the *tutorial1.fpu*:

```
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 (π) 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 pi (π) times radius squared. The **`POWER`** function could use to calculate radius to the power of 2, but for squared values it's easier and more efficient to simply multiply the value by itself. Enter the following line to calculate the area:

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

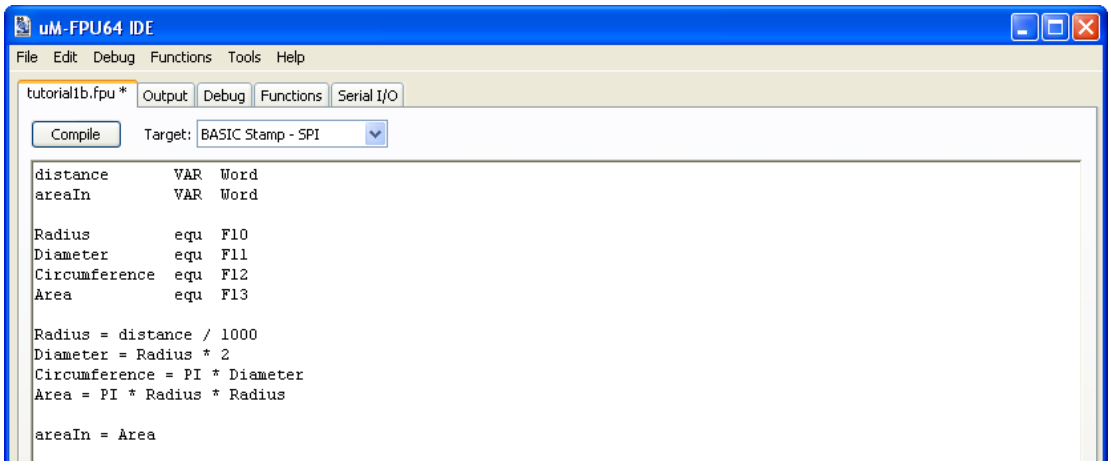
Finally, we'll read the `Area` value back to the microcontroller as a 16-bit integer and print the result. To do this we first add the following definition for the microcontroller variable:

```
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
```

The ***Source Window*** should look as follows:



Click the ***Compile*** button.

Copy Revised Code to the Main Program

Copy the generated code from the IDE Output Window and paste over the previous code in the BASIC Stamp program. Add additional DEBUG statements (as described above) to print the new results.

Copy the *uM-FPU Register Definitions* and *Variable Definitions* from the ***Output Window*** and paste in the Basic Stamp program in the *main definitions* section (replacing the previous definitions).

Copy the Generated Code from the ***Output Window*** and paste in the Basic Stamp program after the *Main* label (replacing the previous code).

Add `DEBUG` and `Print_FloatFormat` statements for each of the calculated values `Radius`, `Diameter`, `Circumference` and `Area`. We'll use the `Print_FloatFormat` with `format = 63` to display the floating point values in a field six characters wide with digits to the right of the decimal point.

```
   DEBUG CR, "Radius:          "
   format = 63
   GOSUB Print_FloatFormat
```

The main section of your BASIC Stamp program should look as follows:

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

'-------------------- uM-FPU Register Definitions ----------------------------
Radius          CON     10              ' uM-FPU register
Diameter        CON     11              ' uM-FPU register
Circumference   CON     12              ' uM-FPU register
Area            CON     13              ' uM-FPU register

'-------------------- 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 Revised 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 FPU source file, replace:

```
areaIn = Area
```

with:

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

Compile the 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:

Saving the Source File

Use the ***File >Save*** command to save the file.

This completes the tutorial on compiling code for the uM-FPU64 chip. 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 FPU Code

This tutorial takes you through some examples of debugging FPU code using the uM-FPU64 IDE. We will use the Basic Stamp program created in the previous tutorial for debugging.

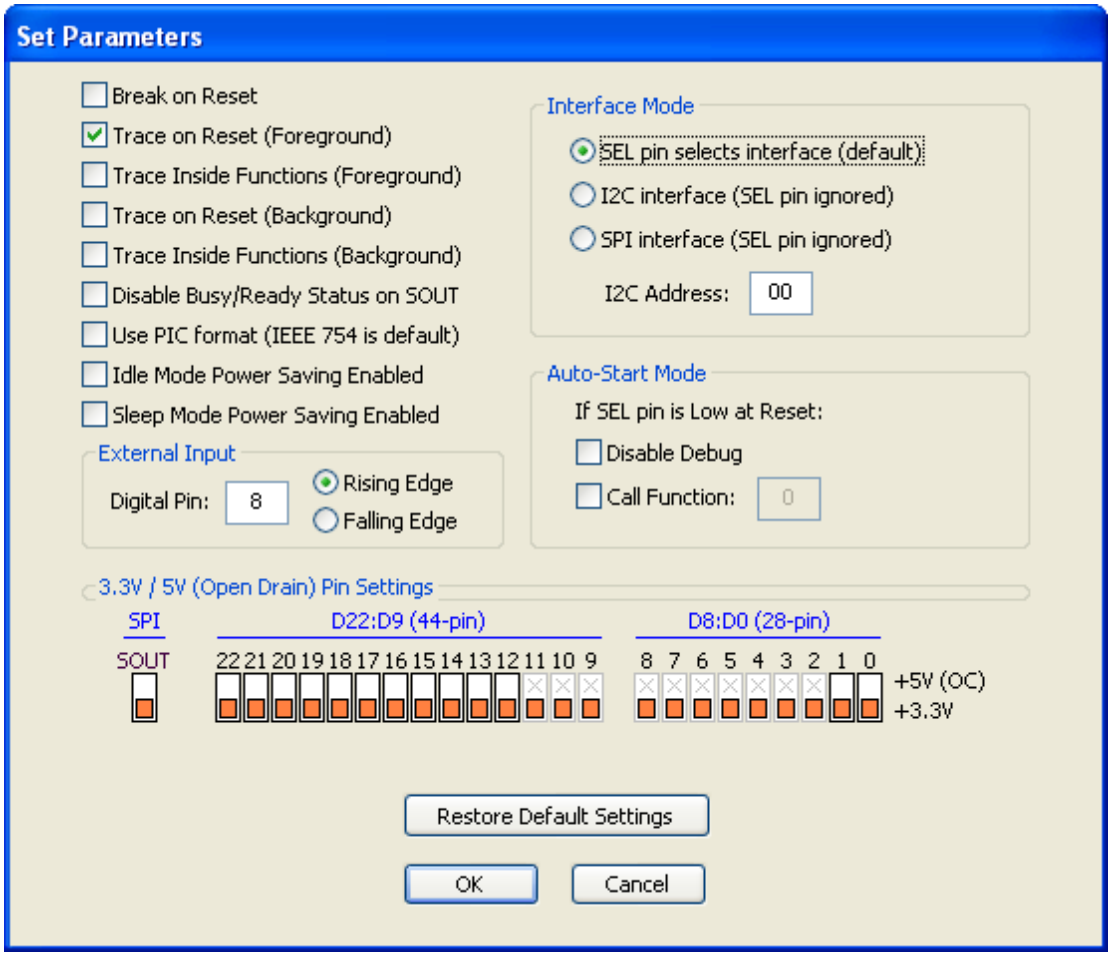
Making the Connection

For debugging, the uM-FPU64 IDE must have a serial connection to the uM-FPU64 chip. Refer to the section at the start of this document called *Connecting to the uM-FPU64 chip*.

Tracing Instructions

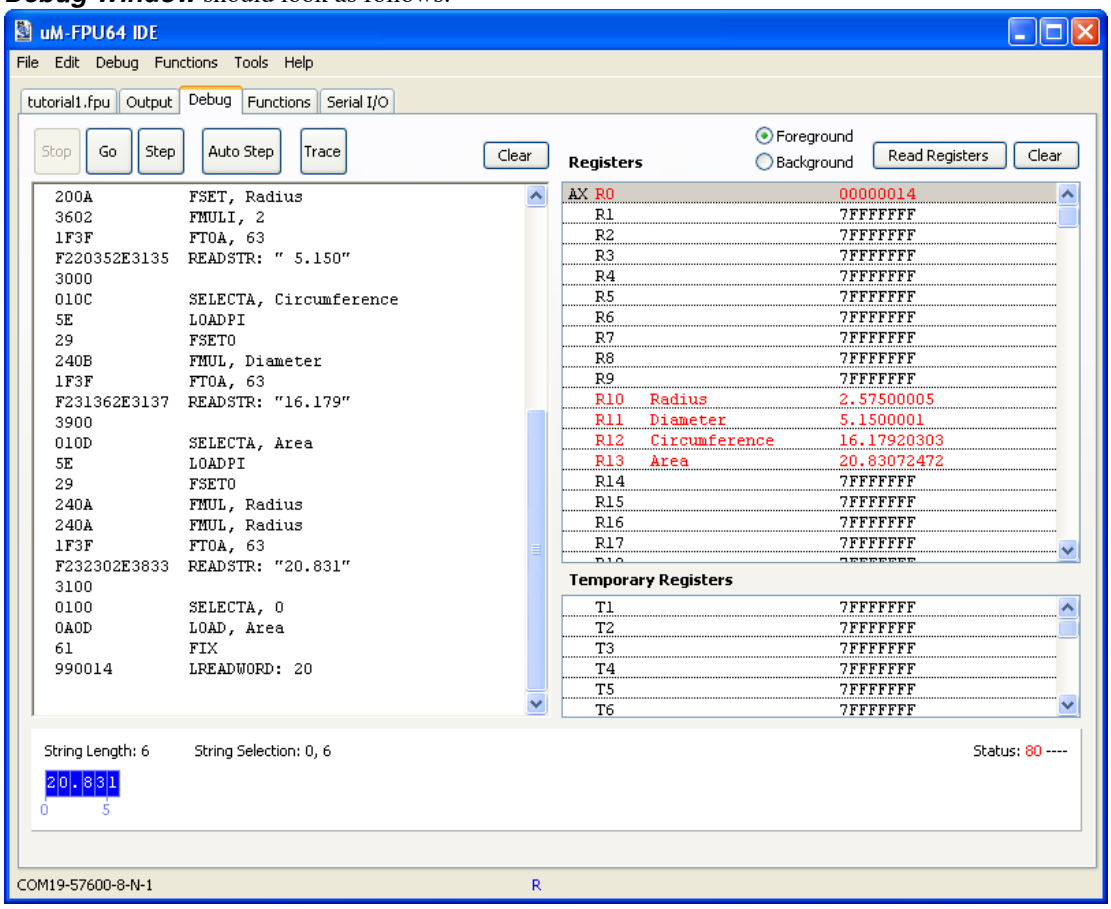
The ***Debug Window*** of the IDE can display a trace of all instructions as they are executed. By default, tracing is disabled. It can be enabled at Reset by setting the ***Trace on Reset (Foreground)*** option in the ***Functions>Set Parameters...*** dialog, or it can be turned on or off at any time by sending the **`TRACEON`** or **`TRACEOFF`** instruction.

For this tutorial we will use the ***Trace on Reset (Foreground)*** option. Select the ***Functions>Set Parameters...*** menu item, and enable the ***Trace on Reset (Foreground)*** option as shown below.



Select the ***Debug Window***, and click the ***Clear*** button above the ***Debug Trace*** to clear the trace area. Now run the *tutorial1.bs2* program that you developed in the previous tutorial. An instruction trace will be displayed in the ***Debug Trace*** area. After the program stops running, click the ***Read Registers*** button to update the ***Register Display, String Buffer, and Status***. Scroll up to the beginning of the ***Debug Trace.***

The ***Debug Window*** should look as follows:



The reset message is displayed at the top of the screen. Every time the FPU resets, a reset message is displayed with a time stamp. The instruction trace shows the hexadecimal bytes of the instruction on the left, followed by the disassembled instruction. If a source file has been compiled with symbol definitions, these symbols are used when displaying the instructions. For instructions that read data from the FPU, the trace will also display the data being sent.

Compare the instructions in the ***Debug Trace*** to the *tutorial1.bs2* program. Tracing is very useful for checking the actual sequence of instruction executed by the FPU. Many programming errors can often be found simply by examining the trace.

Breakpoints

A breakpoint stops execution of FPU instructions. A BREAK message is displayed in the ***Debug Trace*** and the ***Register Display, String Buffer, and Status*** are automatically updated. This enables you to examine the state of the FPU at that point, and then continue execution, or to single step through the code one instruction at a time.

To experiment with breakpoints, add the following statement to the *tutorial1.bs2* program immediately after the *Main* label.

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

Run the *tutorial1.bs2* program again. A breakpoint occurs immediately after printing the version string. By examining the ***Debug Window*** you can see the following:

- the debug trace shows the Reset message and a trace for all previously executed instructions
- the debug trace shows the BREAK message in red
- the version string is displayed in the string buffer
- the AX beside register 0 shows that it's currently selected as register A and register X
- register 0 is displayed in red to indicate it has a new value
- the value in register 0 is the version code
- all other registers are NaN (Not-a-Number)

Single Stepping

By single stepping through the FPU code you can see exactly what's happening. The following example steps through a few instructions.

Click the ***Step*** button (or type the Enter button) to single step. The ***Debug Window*** will change as follows:

- the debug trace shows the **SELECTA,Radius** instruction and the BREAK message
- the A beside register 10 shows that it's now selected as register A
- register 0 is displayed in black since it hasn't changed since the last breakpoint
- 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.

Click the ***Step*** button (or type the Enter button) to single step. The ***Debug Window*** will change as follows:

- the debug trace shows the **LOADWORD,2575** instruction and the BREAK message
- the A beside register 10 shows that it's now selected as register A
- register 0 is displayed in red since it has a new value
- the value in register 0 is 2575.0

Click the ***Step*** button (or type the Enter button) to single step. The ***Debug Window*** will change as follows:

- the debug trace shows the **FSETO** instruction and the BREAK message
- register 0 is displayed in black since it hasn't changed since the last breakpoint
- register 10 is displayed in red since it has a new value
- the value in register 10 is 2575.0

To continue normal execution, click the ***Go*** button.

You can experiment further by moving the **BREAK** instruction to another point in your program, or by adding multiple breakpoints. More advanced single step capabilities are available using the ***Auto Step*** button. See the section entitled *Reference Guide: Debugging uM-FPU64 Code* for more information.

This completes the tutorial on debugging uM-FPU64 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: Programming FPU Flash Memory

User-defined functions and parameter bytes can be programmed in Flash memory on the uM-FPU64 chip. This tutorial takes you through an example of creating some user-defined functions.

Making the Connection

For programming Flash memory, the uM-FPU64 IDE must have a serial connection to the uM-FPU64 chip. Refer to the section at the start of this document called *Connecting to the uM-FPU64 chip*.

Defining functions

In the previous tutorials we developed and tested code to calculate the diameter, circumference, and area of a circle. For this demonstration, we'll define each of these calculations as a separate function.

The **`#FUNCTION`** directive is used to define a function. It specifies the number of the function (0 to 63) and an optional name.

```
#FUNCTION 1 GetDiameter
```

All code that appears after a **`#FUNCTION`** directive will be stored in that function, until the next **`#FUNCTION`** directive, an **`#END`** directive, or the end of the source file. There's an implicit **`RET`** instruction at the end of all functions.

Functions can call other functions. To ensure that the function being called is already defined, function prototypes can be included at the start of the program. Function prototypes are defined using the **`FUNC`** operator, which assigns a symbol name to a function number. We'll use function prototypes in this tutorial example. The following function prototype defines `GetDiameter` as function number 1.

```
GetDiameter         func   1
```

You can assign the function number explicitly, or use the **`%`** character to assign the next unused function number.

```
GetDiameter         func   1
GetCircumference    func   %
GetArea             func   %
```

If a function prototype has been defined, the **`#FUNCTION`** directive just uses pre-defined name.

```
#FUNCTION GetDiameter
```

Calling Functions

Functions are called by entering an ampersand (**`@`**) before the function name or number in the source code.
e.g.

```
@GetDiameter
```

Modifying the Code for Functions

Open the source file called *tutorial1.fpu* that you saved in the first tutorial. Add a function prototype for the three functions called `GetDiameter`, `GetCircumference`, and `GetArea`. 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    ' Microcontroller variable definitions
areaIn              VAR   Word

Radius              equ   F10     ' FPU register definitions
Diameter            equ   F11
Circumference       equ   F12
Area                equ   F13

GetDiameter         func  1       ' Function prototypes
GetCircumference    func  %
GetArea             func  %

#function GetDiameter             ' Function 1
Diameter = Radius * 2

#function GetCircumference        ' Function 2
Circumference = PI * Diameter

#function GetArea                 ' Function 3
Area = PI * Radius * Radius
#end

Radius = FLOAT(distance) / 1000   ' Calculations

@GetDiameter

@GetCircumference

@GetArea

areaIn = ROUND(area)
```

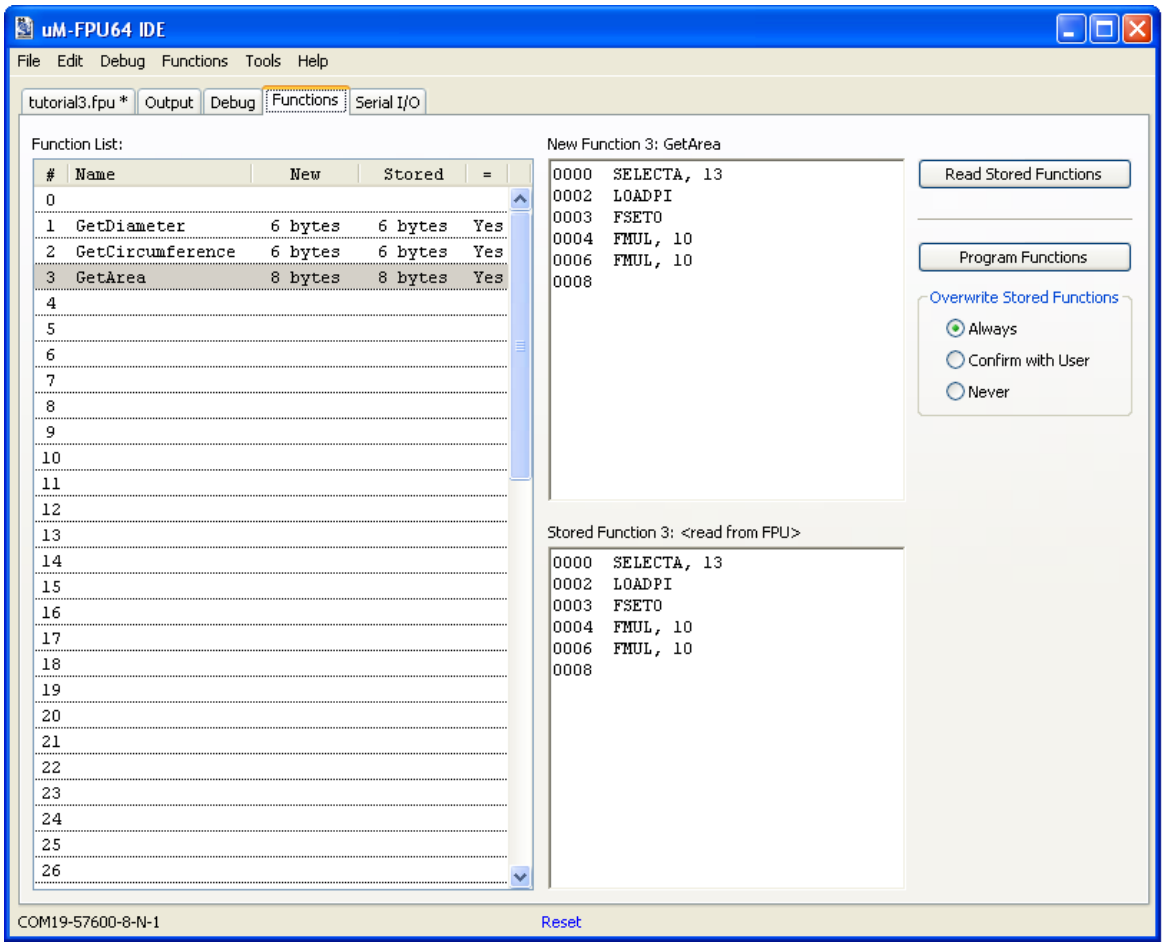
Save the file as *tutorial3.fp4.*

Compile and Review the Functions

Click the ***Compile*** button. In the ***Output Window***, the function code is displayed as comments that show the uM-FPU assembler code that was generated. This is the code that will be programmed to the FPU.

```
' #function GetDiameter
' Diameter = Radius * 2
'      SELECTA, 11
'      FSET, 10
'      FMULI, 2
```

The ***Functions Window*** should look as follows:



The ***Function List*** shows that three functions have been defined. The ***New Function Code*** displays the FPU instructions for the selected function. The ***Stored Function Code*** displays the FPU instructions for the function stored on the FPU. If no function has previously been programmed, the ***Stored Function Code*** will be empty. You can see the code for a different function by selecting it in the ***Function List***.

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 program the functions into Flash memory on the FPU. A status dialog will be displayed as the functions are being programmed. If an error occurs, check the connection. You may need to power the uM-FPU64 chip off and then back on to ensure that it has been reset properly before trying again.

Running the Program

Copy the generated code from the ***Output Window*** 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     10              ' uM-FPU register
Diameter        CON     11              ' uM-FPU register
Circumference   CON     12              ' uM-FPU register
Area            CON     13              ' uM-FPU register

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

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

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

Reset:
  DEBUG CR, "umfpu64-spi", CR
  GOSUB Fpu_Reset                       ' reset the FPU hardware

  IF status <> SyncChar THEN            ' check for synchronization
    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       ' Calculations
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [SELECTA, Radius,
    LONGWORD, distance.HIGHBYTE, distance.LOWBYTE, LSET0, FLOAT,
    LOADWORD, $03, $E8, FDIV0]
  DEBUG CR, "Radius:        "
  format = 63
  GOSUB Print_FloatFormat

  ' @GetDiameter
  SHIFTOUT FpuOut, FpuClk, MSBFIRST, [FCALL, GetDiameter]
  DEBUG CR, "Diameter:      "
  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

Done:
  DEBUG CR, "Done.", CR
  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:

```
uM-FPU64 r401b4

Radius:          2.575
Diameter:        5.150
Circumference:  16.179
Area:           20.831
AreaIn:         21
Done.
```

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

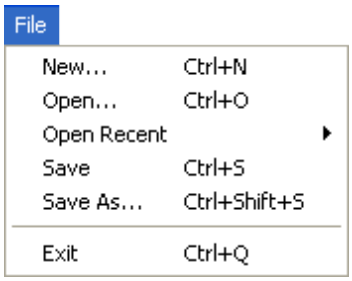
```
uM-FPU64 r401b4

Radius:          2.575
Diameter:        2.575
Circumference:   2.575
Area:            2.575
AreaIn:         65535
Done.
```

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 be able to use the IDE to define your own functions and program them to Flash on the uM-FPU64 chip.

Reference Guide: Menus and Dialogs

File Menu



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.

Open… menu item opens an existing source file, using the file open dialog. 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.

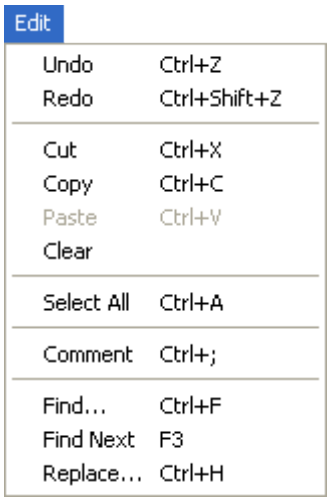
Open Recent menu item provides a sub-menu that lists up to ten source files that were recently saved. Selecting a source file from the sub-menu will open the file. If a previous source file is open and has been changed since the last time it was saved, you will first be prompted to save the previous source file.

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

Save As… menu item displays a file save dialog and allows a new filename to be specified.

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.

Edit Menu



Undo menu item cancels the last edit in the **Source Window**.

Redo menu item restores the edit cancelled by the last **Undo**.

Cut menu item removes the selected text from the ***Source Window***.

Copy menu item copies the selected text from the ***Source Window*** to the clipboard.

Paste menu item pastes the text in the clipboard to the current selection point in the ***Source Window***.

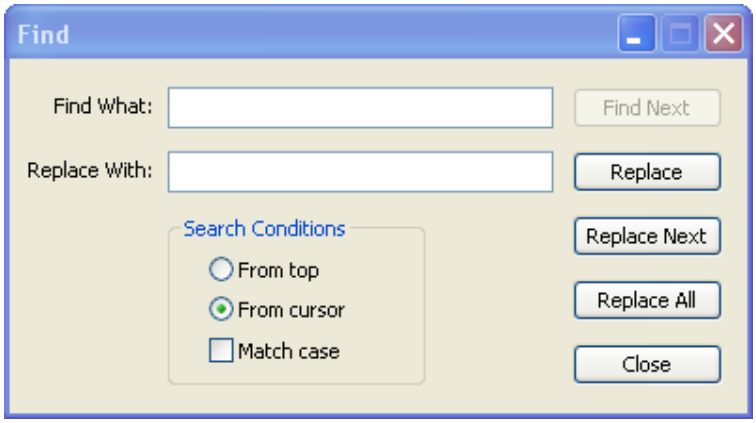
Clear menu item deletes the selected text from the ***Source Window***.

Select All menu item selects all of the text in the current text field.

Comment menu item is used to add a semi-colon as the first character of every currently selected line in the ***Source Window***. This provides a way to quickly comment out a block of code. If all of the lines currently selected have a semi-colon as the first character, the menu item changes to **Uncomment**.

Uncomment menu item removes the semi-colon from the start of all selected lines.

Find... menu item brings up the **Find** Dialog.



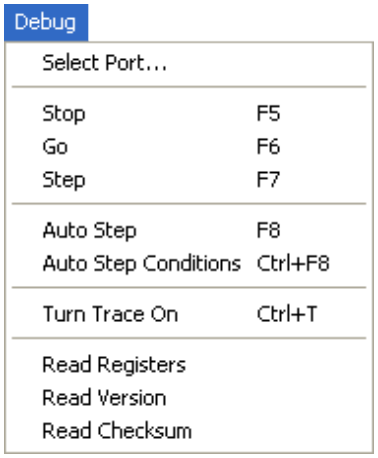
The **Find** dialog is a moveable dialog and can be placed alongside the ***Source Window*** and left open when multiple find and replace operations are done. The **Find What** field specified the string to search for, and the **Replace With** field specifies the string to replace it with. If the **From top** search condition is selected, the search starts from the top of the window. The search condition will automatically change to **From cursor** on the first successful match. If the **From cursor** search conditions is selected, the search starts from the current cursor position. When the **Match case** option is selected, the search is case sensitive. The following special characters can be used in the Find or Replace strings: **`\t`** for a tab character, **`\r`** for end of line, and **`\\`** for backslash.

The **Find Next** button searches the ***Source Window*** for the next match. The **Replace** button replaces the matched string. The matching text is highlighted on the first button press and replaced by the **Replace With** string on the next button press. The **Replace All** button replaces all occurrences of the **Find What** string with the **Replace With** string. The **Close** button closes the **Find** dialog.

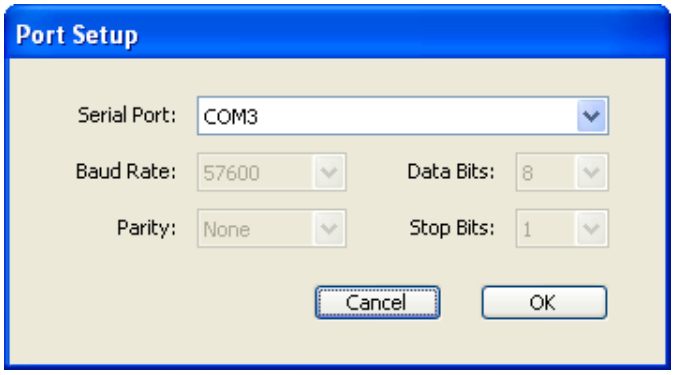
Find Next menu item finds the next match based on the current search conditions in the **Find** dialog.

Replace menu item brings up the **Find** Dialog.

Debug Menu



Select Port... menu item is used to display the **Port Setup** dialog which is used to select the serial communications port.



Go, **Stop**, and **Step** menu items have the same function as the **Go**, **Stop** and **Step** buttons in the ***Debug Window***.

Turn Trace On and **Turn Trace Off** menu items have the same function as the **Trace** button in the ***Debug Window***.

Auto Step Conditions menu item brings up the **Auto Step Conditions** dialog. See the section entitled *Reference Guide: Auto Step and Conditional Breakpoints* for more details.

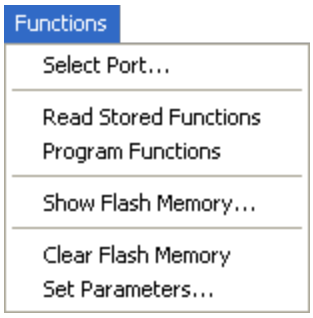
Auto Step menu item continues execution in auto step mode. See the section entitled *Reference Guide: Auto Step and Conditional Breakpoints* for more details.

Read Registers menu item has the same function as the **Read Registers** button in the ***Debug Window***.

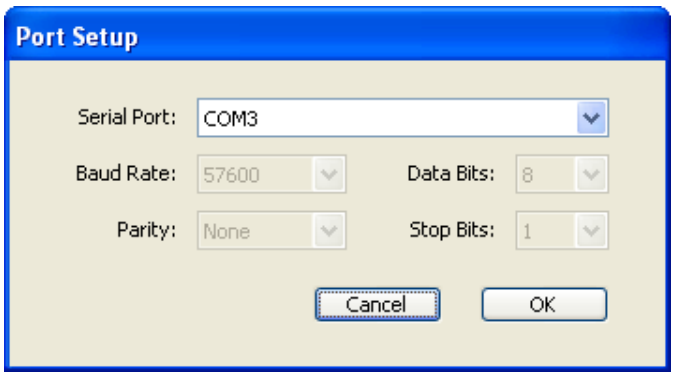
Read Version menu item will display the version of the FPU in the ***Debug Trace***.

Read Checksum menu item will display the checksum of the FPU in the ***Debug Trace***.

Functions Menu



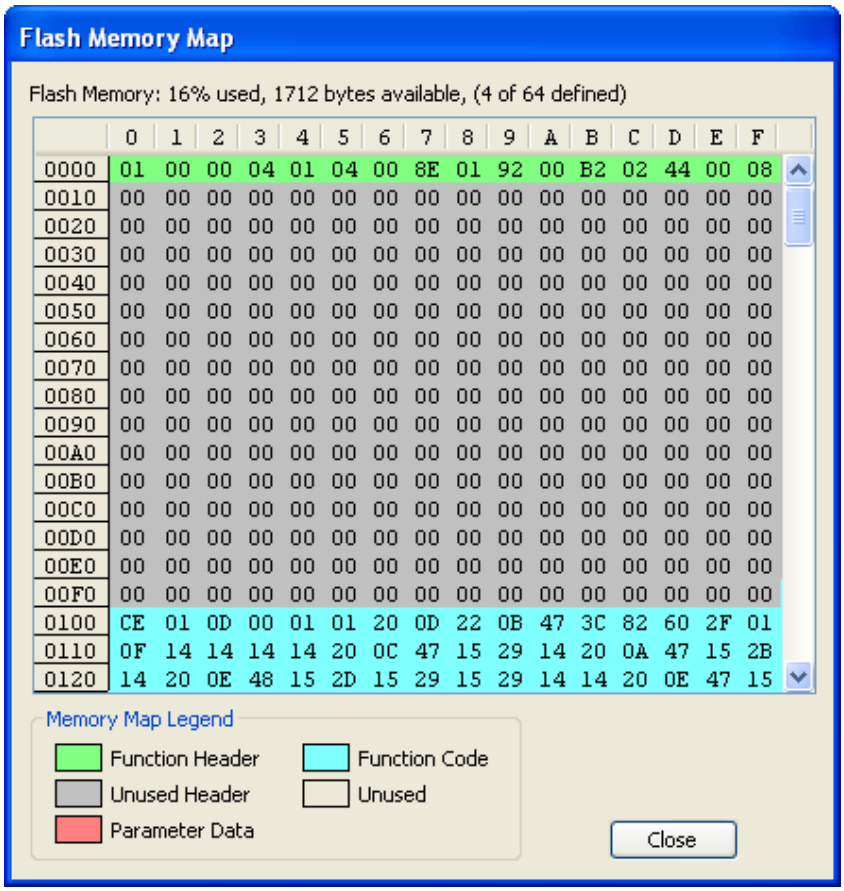
Select Port... menu item is used to display the **Port Setup** dialog which is used to select the serial communications port.



Read Stored Functions menu item has the same function as the **Read Stored Functions** button. It reads the flash memory and updates the function list in the ***Function Window***.

Program Functions menu item has the same function as the **Program Functions** button. It programs the user-defined functions to the FPU chip.

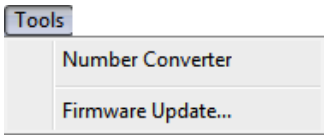
Show Flash Memory... menu item displays a memory map showing the usage of the Flash memory reserved for user-defined functions on the uM-FPU64 chip. A status line at the top shows the percent of memory used and the number of bytes available.



Clear Flash Memory menu item will clear all of the user-defined functions from Flash memory on the uM-FPU64 chip. A dialog will be displayed requesting confirmation before the functions are cleared from memory.

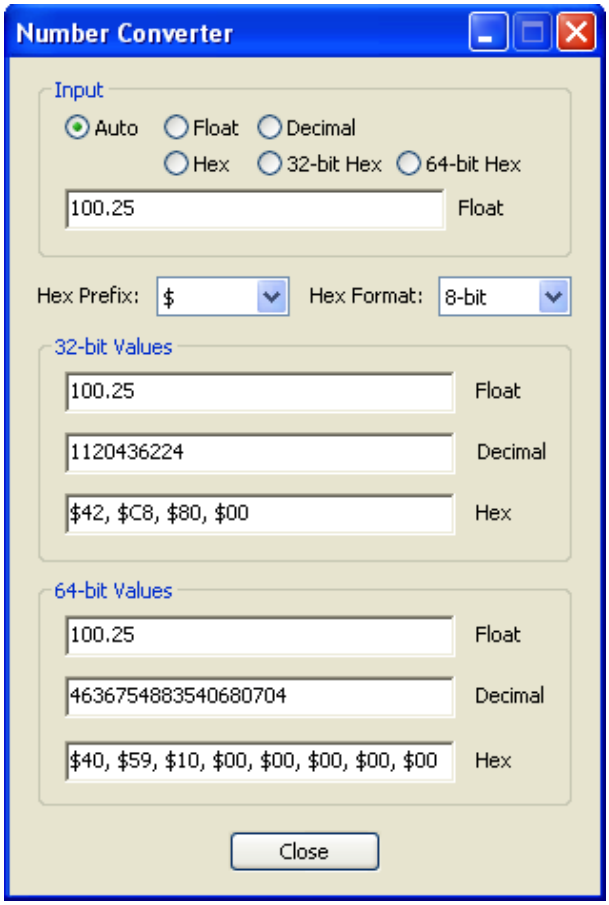
Set Parameters… menu item is used to set the FPU parameter bytes. See the section entitled *Reference Guide: Setting uM-FPU64 Parameters* for more details.

Tools Menu



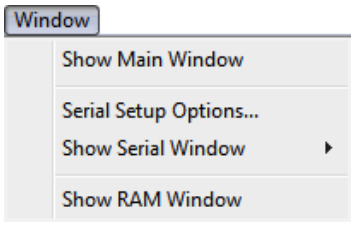
Number Converter menu item is used to bring the **Number Converter** window to the front. The number converter provides a quick way to convert numbers between various 32-bit and 64-bit formats. Floating point, decimal and hexadecimal numbers are supported. The **Auto**, **Float**, **Decimal**, and **Hexadecimal** buttons above the **Input** field determine how the input is interpreted. If **Auto** is selected, the input type is determined automatically based on the characters entered in the **Input** field. The input type is displayed to the right of the **Input** field. The input type can be manually set using the **Float**, **Decimal** and **Hexadecimal** buttons. Invalid characters for the selected type are displayed in red, and will be ignored by the converter. The **Output** fields display the input value in all three formats. The hexadecimal value can be displayed in 8-bit, 16-bit, 32-bit, or 64-bit format, with a choice of prefix characters. The format can be selected to match the format used by microcontroller programs.

One of the handiest ways of using the number converter is with copy and paste. You can copy a number from program code or a trace listing, and paste into the **Input** field. The **Input** field accepts floating point numbers, decimal numbers, and hexadecimal numbers in 8-bit, 16-bit, 32-bit, and 64-bit formats. You can copy from the **Output** fields to program code.



Firmware Update... menu item is used to update the uM-FPU64 firmware.

Window Menu

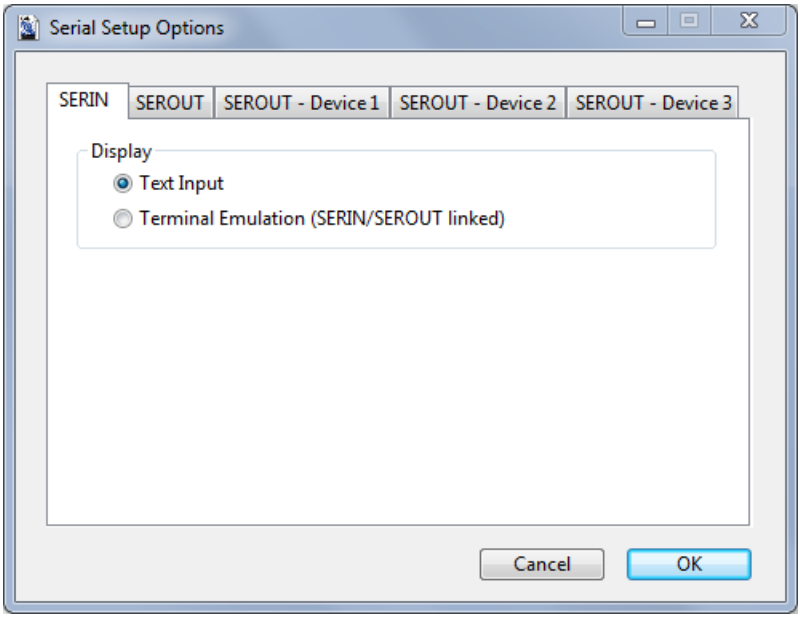


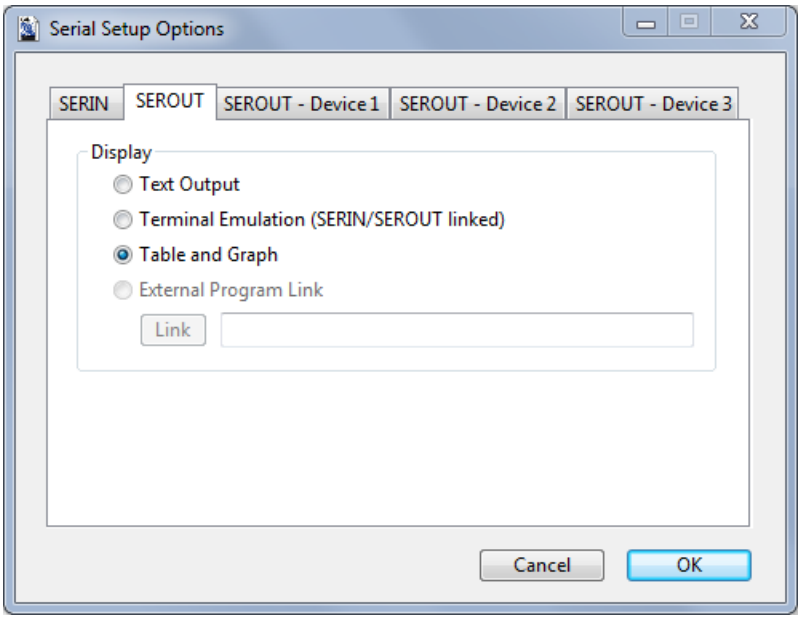
Show Main Window

Brings the main IDE window to the front.

Serial Setup Options…

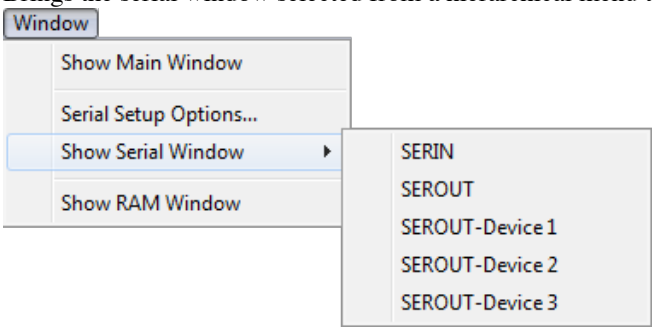
Displays a tabbed dialog that is used to set the display type for each of the serial windows.





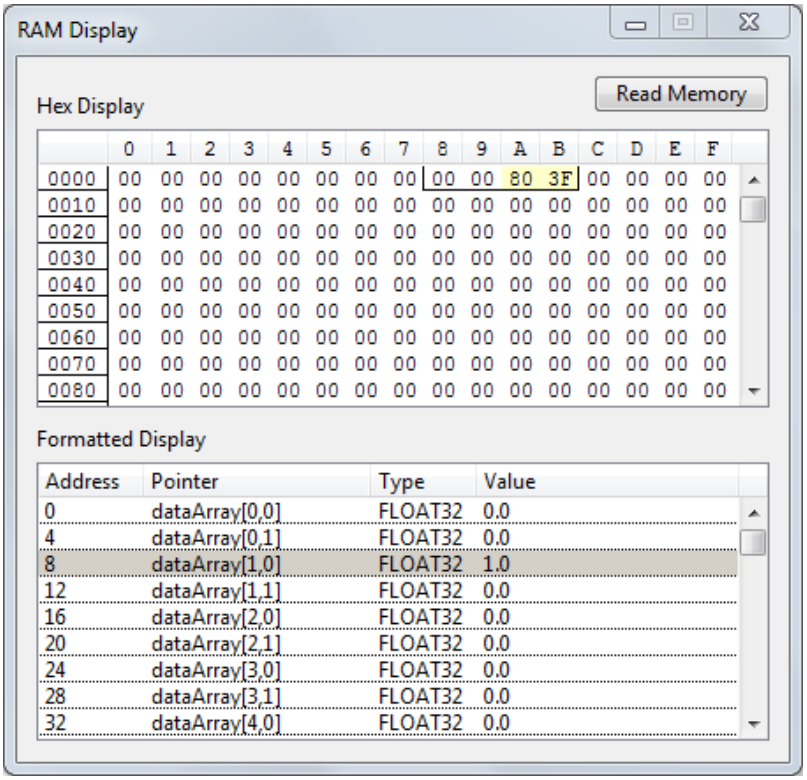
Show Serial Window

Brings the serial window selected from a hierarchical menu to the front.



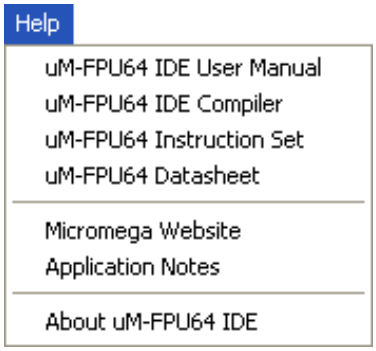
Show RAM Window

The **RAM Display** window is brought to the front. This window is used to view the contents of RAM.



- a hex display is shown at the top
- a formatted display is shown at the bottom
- a box is drawn around the selected item in the hex display
- non-zero items are highlighted in both displays
- a pop-up menu is available in the formatted display to change the format of the displayed values
- selecting in one display, selects the corresponding item in the other display
- memory pointers identified in compiler are loaded from the Debug Registers
- registers are loaded automatically on Break, but can also be loaded manually with *Read Registers* button
- the *Read Memory* button must be pressed to read current RAM contents
- if a memory pointer is defined as an array, the array index names are shown
- data items at pointers or pointer arrays are automatically formatted according to the pointer type

Help Menu



uM-FPU64 IDE User Manual, **uM-FPU64 IDE Compiler, uM-FPU64 Instruction Set**, and **uM-FPU64 Datasheet** menu items display documentation files using the default PDF viewer. The IDE will open the files on the Micromega website using the default web browser.

Micromega Website menu item opens the Micromega website using the default web browser.

Application Notes menu item opens the application notes page on the Micromega website using the default web browser.

About uM-FPU64 IDE menu item displays a dialog with product identification, release version and release date of the uM-FPU64 IDE software. A link to the Micromega website is also provided

Reference Guide: Compiler and Assembler

The uM-FPU64 IDE provides a compiler for generating uM-FPU64 code for either a target microcontroller, or for user-defined functions that are stored in Flash memory on the FPU. The ***Source Window*** has a built-in editor for entering the source code. The source code contains symbol definitions and math equations that will be converted to FPU instructions by the compiler. The output format is customized to the correct syntax for the target microcontroller. User-defined functions can be programmed to Flash memory on the uM-FPU64 chip.
Symbol definitions can include FPU registers, variables, and constants. Math equations can use long integer or floating point values, and can contain defined symbols, math operators, functions and parentheses. The compiler also supports an in-line assembler for entering FPU instructions directly.

See the *uM-FPU64 IDE Compiler* document for a description of the compiler and assembler.

Reference Guide: Debugger

Utilizing the built-in debug monitor on the uM-FPU64 chip, the IDE provides a high-level interface for debugging programs that use the uM-FPU64 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.

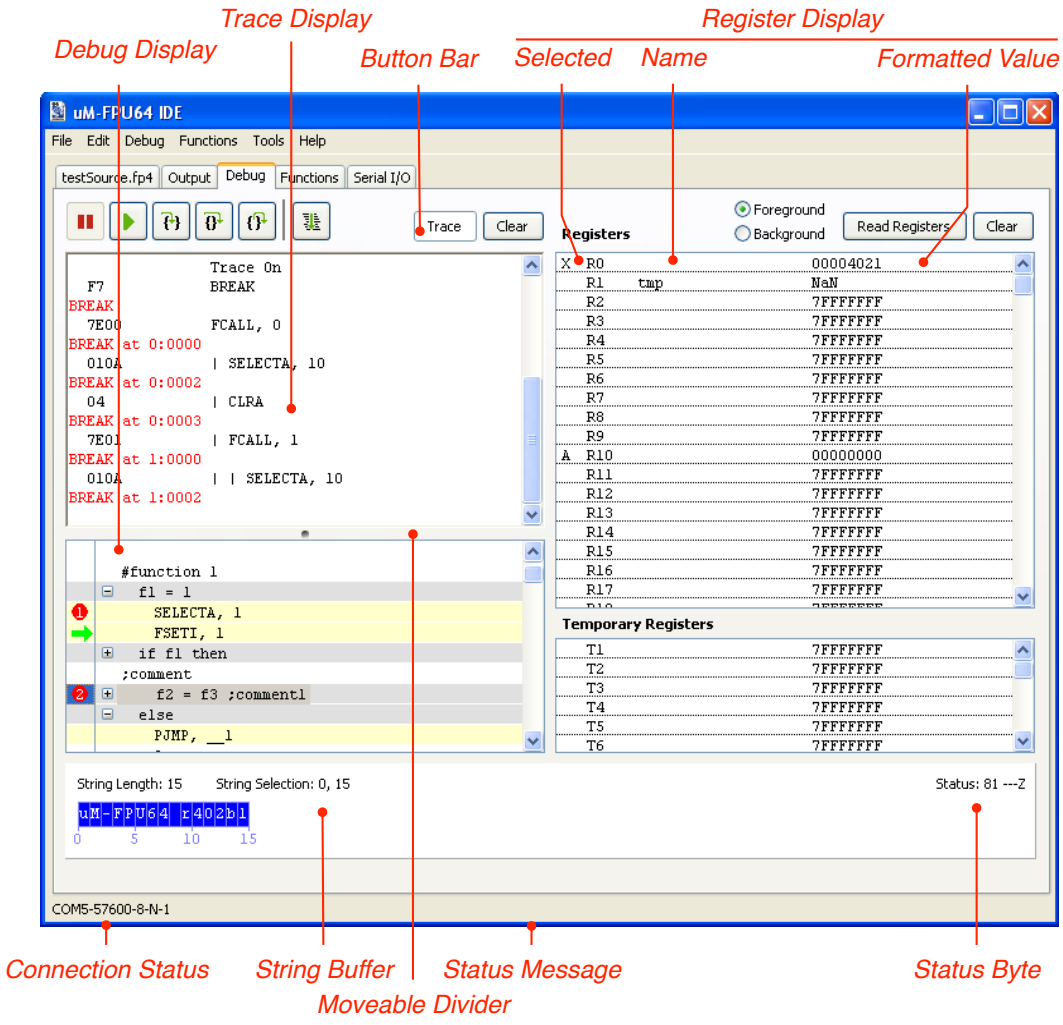
Making the Connection

For debugging, the uM-FPU64 IDE must have a serial connection to the uM-FPU64 chip. Refer to the section at the start of this document called *Connecting to the uM-FPU64 chip*.

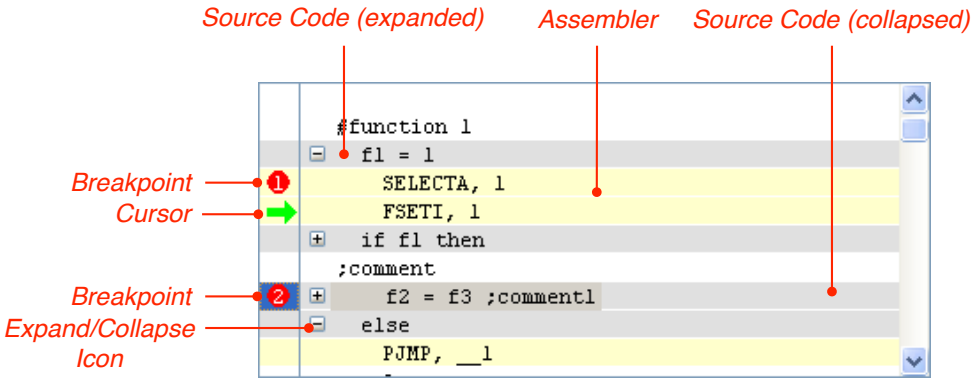
Source Level Debugging

Source level debugging is only available for user-defined functions. The source file is displayed below the trace display. A movable divider is located between the trace display and debug display. Breakpoints can be set on any executable line shown in the debug display (both source level and assembler). All executable lines have an expand/collapse icon. Source lines can be expanded to display the assembler code generated by the source line. When the debugger is active, a cursor shows the next instruction to be executed. If a source line is expanded, the debugger will step by assembler instruction. If the source line is collapsed, the debugger will step by source line.

Debug Window



Source-level Debug Display

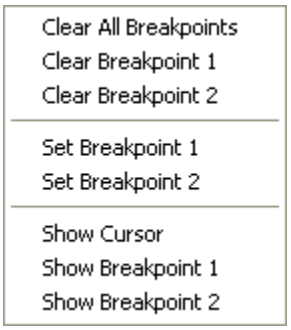


Left column: Breakpoint and cursor display.
Right column: Source code and assembler code display.

White background: Source code (non-executable).
Gray background: Source code (executable).
Yellow background: Assembler code (executable).

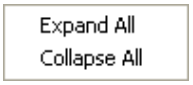
Double-click on left column (executable line):
Sets or clears breakpoint.
If no previous breakpoint, sets the next breakpoint.
If no more breakpoints, displays a placeholder.
If breakpoint or placeholder present, they are cleared.

Right-click on left column:

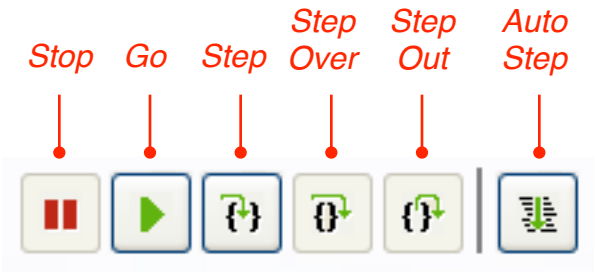


Double-click on right column (executable line):
Expands or collapses the individual line.
Breakpoints are cleared on lines that are collapsed.
Cursor is moved to expanded or collapsed line.

Right-click on right column:



Debug Buttons



Stop
- Stop execution and enter debugger.

Go
- Start or continue execution.

Step
- Step to next executable line.
- If source code is unexpanded, the step is to next executable source line.
- If source line is expanded, the step is to next assembler instruction.

Step Over
- Step to next executable line in the same function (steps over function calls).
- If source code is unexpanded, the step is to next executable source line.
- If source line is expanded, the step is to next assembler instruction.

Step Out
- Steps out of current function.

Auto Step
- Functionality is unchanged from previous version.

The ***Trace Display*** displays messages and instruction traces. The Reset message includes a time stamp, is is displayed whenever a hardware or software reset occurs. Instruction tracing will only occur if tracing is enabled. This can be enabled at Reset by setting the ***Trace on Reset*** option in the ***Functions>Set Parameters...*** dialog, or at any time by sending the **`TRACEON`** instruction.

The ***Register Display*** shows the value of all registers. Register values that have changed since the last update are shown in red. The ***String Buffer*** displays the FPU string buffer and string selection, and the ***Status Byte*** shows the FPU status byte and status bit indicators. The ***Register Display***, ***String Buffer***, and ***Status Byte*** are only updated automatically at breakpoints. They can be updated manually using the ***Read Registers*** button.

The ***Go***, ***Stop***, ***Step*** and ***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.

Trace Display

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.

```
----------------------------------------
RESET: 2011-09-27 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 ***Trace*** button. If tracing is enabled, all FPU instructions are displayed as they are executed. The opcode and data bytes are displayed on the left, and the FPU instructions are displayed on the right in assembler format.

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

The ***Trace*** button toggles the trace mode on and off.

Clicking the ***Clear*** button above the ***Debug Trace*** window will clear the contents of the ***Debug Trace*** window.

Breakpoints

Breakpoints can be inserted into a program using the **BREAK** instruction, or initiated manually with the ***Stop*** button. Breakpoints occur after the next FPU instruction finishes executing. When a breakpoint occurs, the last 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.

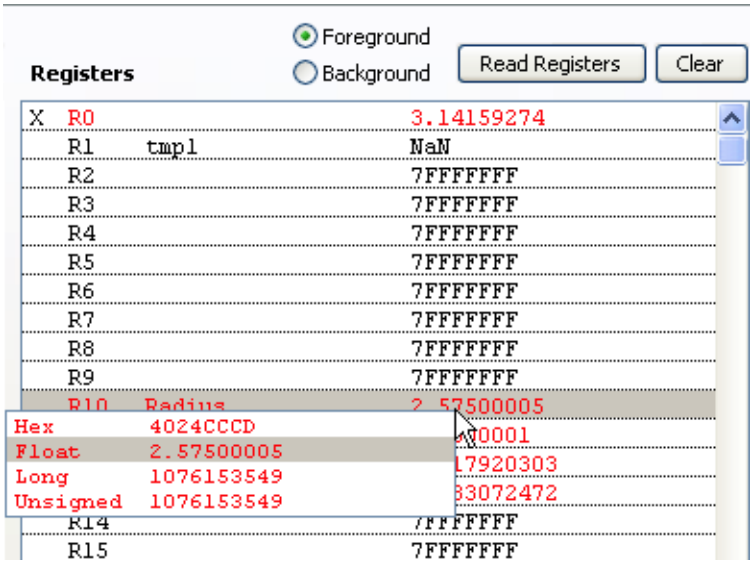
```
 5E           LOADPI
BREAK
```

The ***Go***, ***Stop***, and ***Step*** 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 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 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.

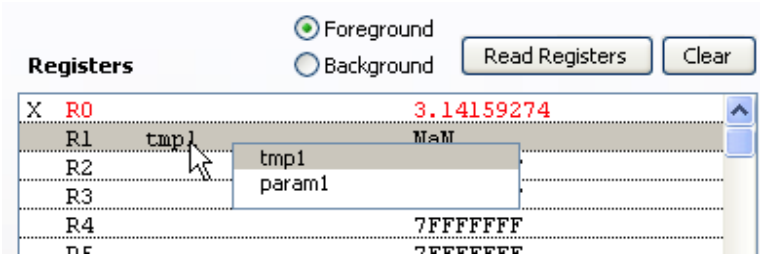
The Register Panel

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.

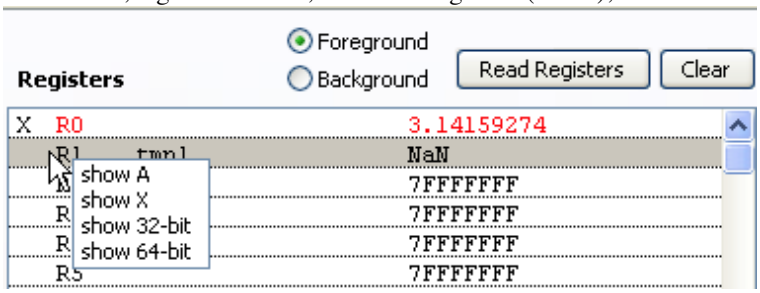
For each register, the register number, optional register name, and formatted value is displayed. If you right-click on the formatted value, a pop-up menu is displayed with the register value displayed in hexadecimal, floating point, long integer, and unsigned long integer format. If you select a different format, the display will be updated to show that format. The format of multiple registers can be changed by selecting a group of registers prior to the right-click for the format pop-up menu.



Register names are automatically set from the register definitions in the source file. Registers can often have several different names assigned. If you right-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.



If you right-click on the register number, a pop-up menu is displayed that always you to scroll the display to the register A value, register X value, the 32-bit registers (0-127), or the 64-bit registers (128-255).



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.

Error messages

<data error>

The IDE communicates with the uM-FPU64 chip using a serial connection. If the IDE detects an error in the data received from the FPU, the data error message is displayed in the ***Debug Trace***. This can sometimes occur

immediately before a Reset, if the reset interrupts a trace operation in progress. This situation can be ignored. If it occurs at other times it indicates a problem with the serial communications. The trace in the ***Serial I/O*** window can be reviewed and may help determine the source of the problem.

<trace suppressed>

In certain circumstances, the FPU is capable of sending data faster than the PC can handle it. If this occurs, the trace suppressed message is displayed, and the IDE attempts to recover by suppressing data, resynchronizing, and continuing. This situation should not normally occur, but can occur if excessive amounts of trace data are being produced such as tracing a user-defined function that is looping. To avoid this situation, the **TRACEOFF** and **TRACEON** instructions can be used to selectively disable tracing.

<trace limit *xx*>

The IDE will retain up to 100,000 characters in the ***Debug Trace***. This is normally more than sufficient for tracing and debugging. The ***Debug Trace*** buffer can be cleared with the ***Clear*** button. If the buffer is exceeded, the first portion will be deleted, and the trace limit message displayed in its place. The trace limit messages are numbered sequentially. This message does not necessarily indicate an error, unless it occurs in conjunction with one of the messages described above.

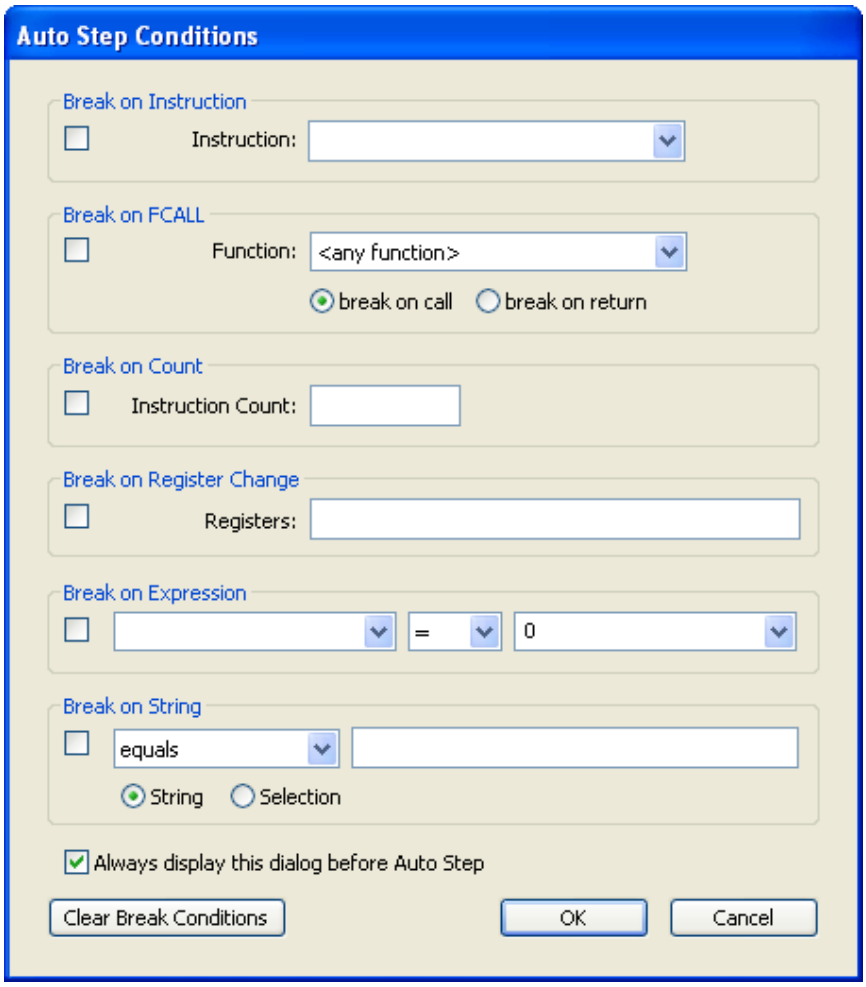
Reference Guide: Auto Step and Conditional Breakpoints

The Auto Step feature provides a means to automatically single step through FPU instructions. This feature, in conjunction with Auto Step Conditions, can be used to implement conditional breakpoints. Conditional breakpoints stop instruction execution when one of the specified conditions occur. Breakpoints can be set for a variety of conditions including: when a particular instruction is executed, when a user-defined functions is called, when a specified number of instructions have been executed, when a register value changes or matches a particular expression, or when a string comparison matches a particular condition. Multiple conditions can be specified, and a breakpoint will occur when any of the conditions is met.

Conditional breakpoints are only active when the ***Auto Step*** operation is used. They are not active when the ***Go*** or ***Step*** operation is used. Instruction execution is much slower using ***Auto Step*** since an internal breakpoint occurs for each instruction, and the debug trace and register data are checked for ***Auto Step Conditions***.

Auto Step is activated by clicking the ***Auto Step*** button, or selecting the ***Debug > Auto Step*** menu item. Auto Step Conditions are set by right-clicking the ***Auto Step*** button, or selecting the ***Debug > Auto Step Conditions*** menu item. The ***Auto Step Conditions*** can also be set to appear each time the ***Auto Step*** button is pressed.

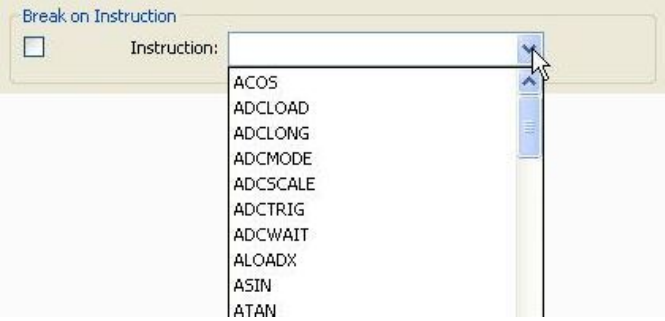
Auto Step Conditions Dialog



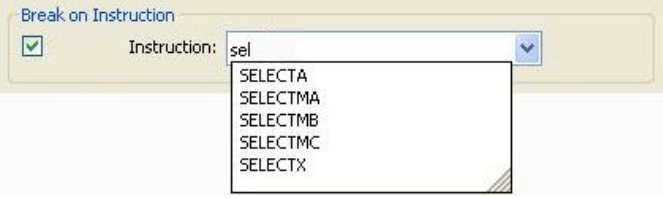
Break on Instruction

This condition causes a breakpoint when a particular instruction is executed. The instruction is specified using assembler format as shown below.

The opcode can be selected from a pop-up menu,

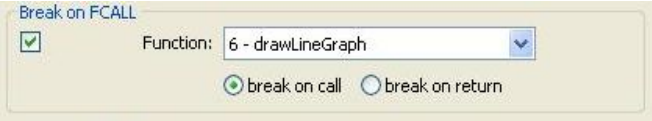


or the opcode can be typed in the field. An auto-complete feature is provided to assist in typing the opcode.

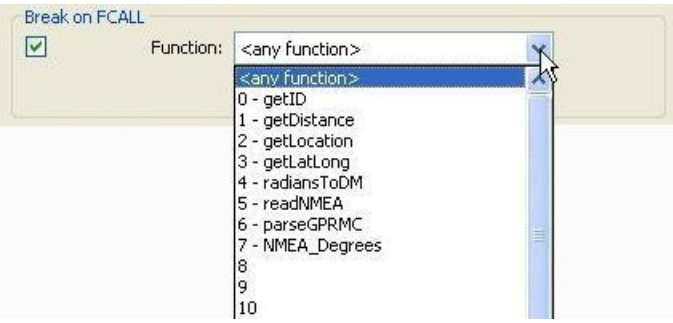


Break on FCALL

This condition causes a breakpoint when a user-defined function is called, or when it returns.

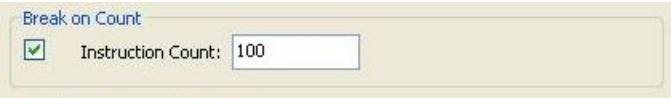


The function is selected from a pop-up menu. The menu has all of the function numbers. If functions have been defined in the current source file, and compiled, the function name is also displayed in the menu. The special item *<any function>* can also be selected to cause a breakpoint on any function call.



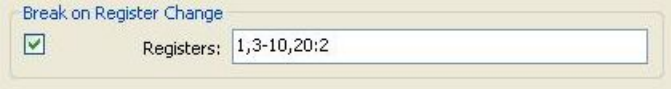
Break on Count

This condition causes a breakpoint after a specified number of instructions has executed.



Break on Register Change

This condition causes a breakpoint when the value changes in one of the specified registers.



Multiple registers can be specified separated by commas. A register can be specified as:

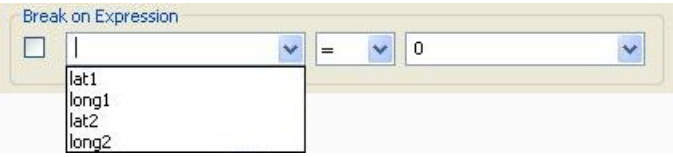
- a single register value (e.g. 1)
- a range of register values (e.g. 3-10 which selects registers 3 through 10)
- an array of register values (e.g. 20:2 which selects two registers starting at registers 20)

If register names have been defined in the current source file, and compiled, the names can also be used.

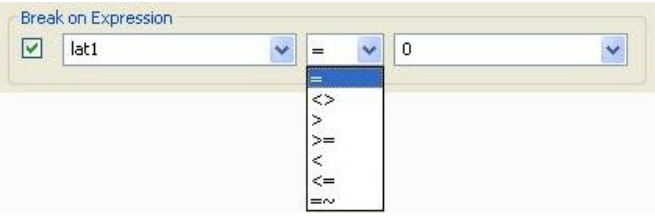
Break on Expression

This condition causes a breakpoint whenever the expression is true.

The left side of the expression must be a register. A register number can be typed in, or if registers have been defined in the current source file, and compiled, a pop-up menu can be used.



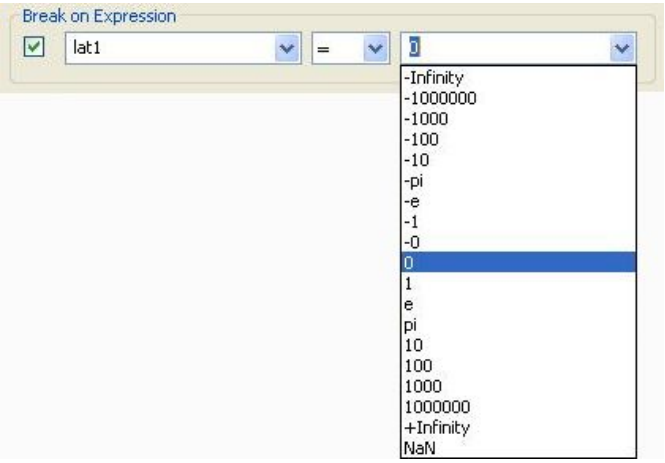
The operator used by the expression is chosen from the middle pop-up menu



The operators are as follows:

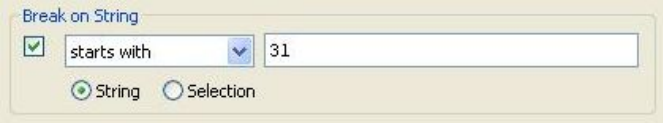
=	equal
<>	not equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
=~	approximately equal

The approximately equal operator is used for floating point values. The condition is true if the register value is greater than (value - 0.000001) and less than (value + 0.000001).
The left side of the expression can be any value. The value can be typed in or the pop-up menu can be used for predefined values.

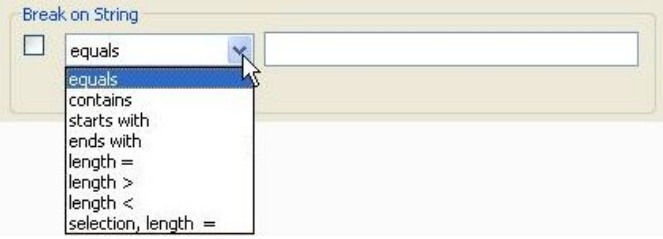


Break on String

This condition causes a breakpoint if the string comparison is true.



The string comparison can either be the entire string buffer, or the current string selection. The comparison operator is selected from the left pop-up menu, and the string to compare is entered in the field on the right.

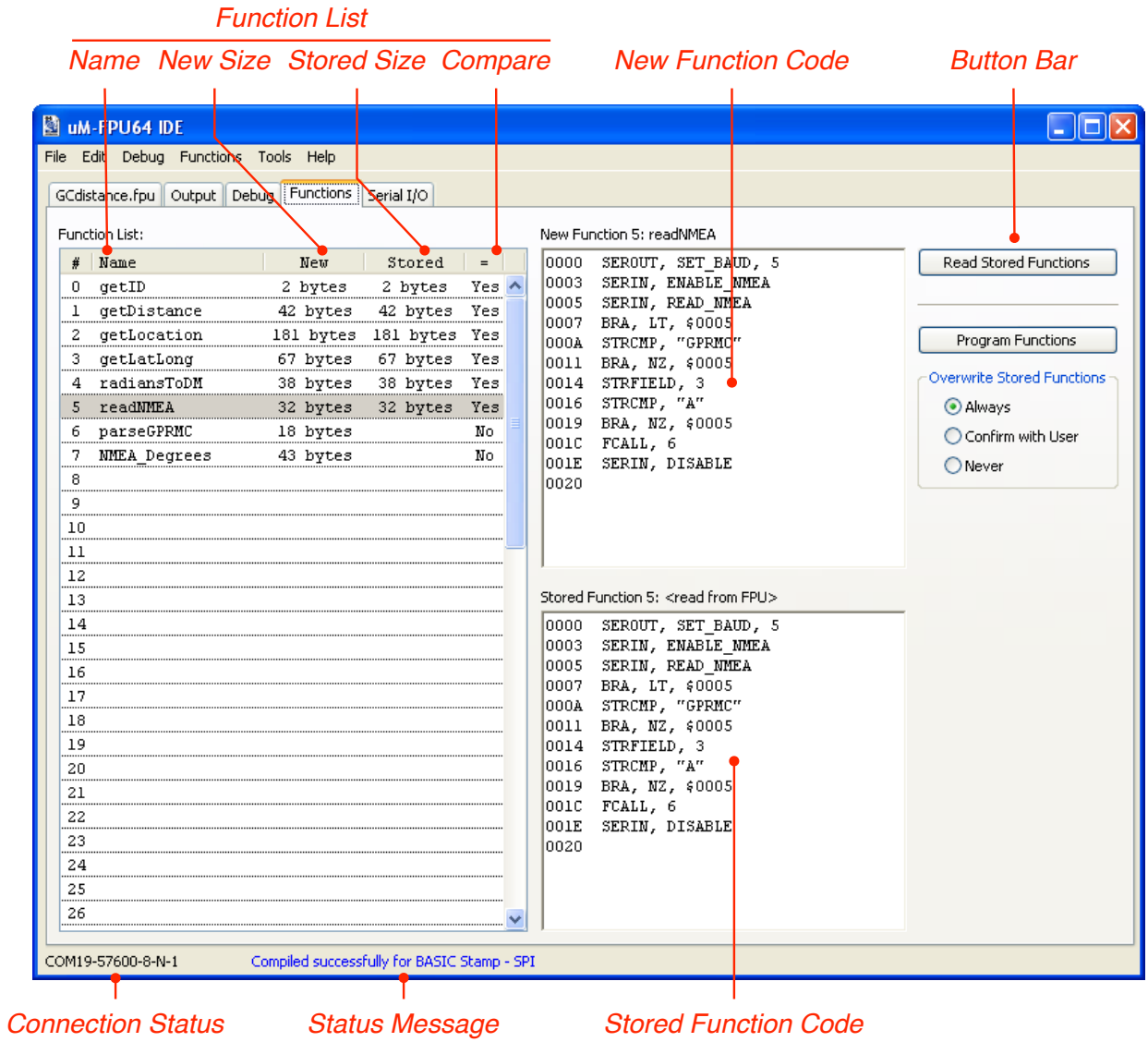


The comparisons for length require a decimal number to be entered in the field on the right. The comparisons for selection, length require two decimal numbers separated by a comma to be entered in the field on the right.

Reference Guide: Programming Flash Memory

The ***Function*** window provides support for storing user-defined functions on the uM-FPU64 chip. Stored functions can reduce memory usage on the microcontroller, simplify the interface and often increase the speed of operation. The uM-FPU64 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 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-FPU64 Code* for more details.

Function Window



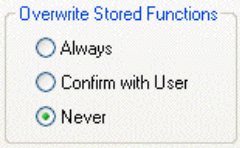
The ***Function List*** provides information about each function defined by the compiler and stored on the FPU. The **Name** column in the ***Function List*** displays the name of all functions defined in the source file. The **New** column shows the size in bytes of the functions defined in the source file, and the **Stored** column displays the size in bytes of functions currently stored on the FPU (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 ***New Function Code*** displays the FPU instructions for compiled functions, and the ***Stored Function Code***

displays the FPU instructions for functions stored on the FPU. The function to be displayed is selected by selecting one of the functions in the ***Function List***.

The ***Read Stored Functions*** button is used to read the functions currently stored on the FPU and update the ***Function List***.

The ***Program Functions*** button is used to program new functions to the uM-FPU64 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.

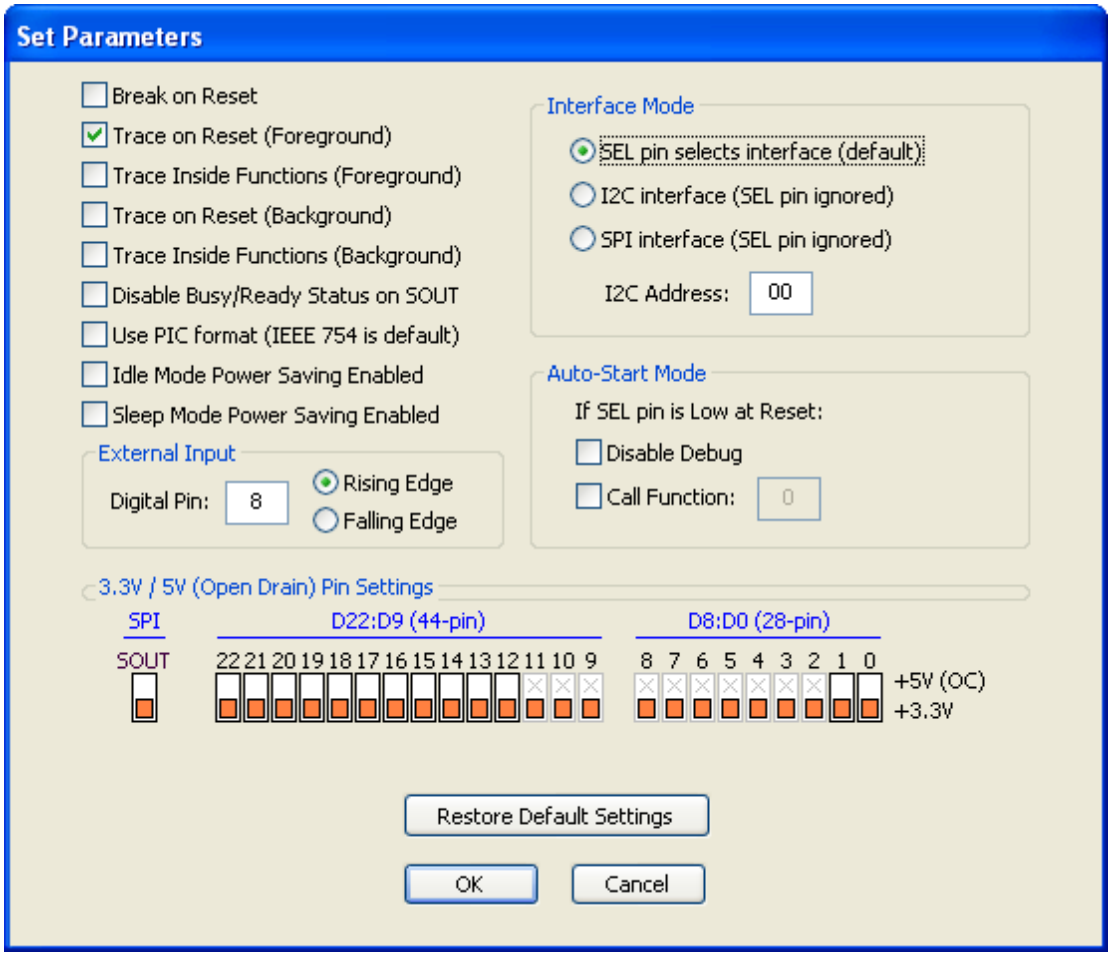
If the **Confirm with User** option is selected, you are asked to confirm whether a new function should replace the previously stored function.

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

Reference Guide: Setting uM-FPU64 Parameters

The **Set Parameters...** menu item is used to set the uM-FPU64 mode parameter bytes.

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 (Foreground)

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

Trace Inside Functions (Foreground)

If this option is selected, debug tracing will be enabled inside functions called by foreground tasks.

Trace on Reset (Background)

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

Trace Inside Functions (Background)

If this option is selected, debug tracing will be enabled inside functions called by background events.

Disable Busy/Ready status on SOUT

If this option is selected, the Busy/Ready status will not be output on the **SOUT** pin, and the **/BUSY** pin must be monitored for the Busy/Ready status.

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-FPU64 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 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 FPU or written to the FPU, 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-FPU64 Instruction Set*.

> 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-FPU64 chip will go into a low power mode when idle.

Sleep Mode Power Saving Enabled

If this option is selected, the uM-FPU64 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

This option selects which digital I/O pin will be used for the external input, and specifies the active edge.

Interface Mode

By default, the **SEL** pin on the uM-FPU64 chip is read at Reset to determine if the SPI or I²C interface is to be used. The interface mode parameter can be used to force selection of SPI or I²C at Reset (ignoring the **SEL** pin).

I2C Address

By default, the I²C address used by the uM-FPU64 chip is `C8` (hexadecimal) or `1100100x` (binary). If the default address conflicts with another I²C device, or if multiple uM-FPU64 chips are used on the same I²C bus, the address can be changed to any other valid I²C 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.

Auto-Start Mode

A user-defined function can be called and Debug Mode can be disabled when the FPU is Reset. If the **Disable Debug** option is selected, Debug Mode will be disabled at Reset. This is useful if the **SERIN** and **SEROUT** pins are being used for other purposes (e.g. GPS input, LCD output) and prevents the `{RESET}` message from being sent to the **SEROUT** pin at Reset. If the **Call Function** option is selected, the specified function will be called at Reset.

These options are only checked if the **CS** pin is Low at Reset. If both the **CS** pin and **SERIN** pin are High at Reset, the auto-start function is not called, and Debug Mode will always be entered. This provides a way to override the auto-start mode once it is set. To use auto-start with an I²C interface, the interface mode bits must be set to I²C (as described above). It's recommended that the interface be set to SPI or I²C using the interface bits whenever auto-start mode is used, so that the **CS** pin can be used to enable or disable the auto-start mode.

3.3V / 5V (Open Drain) Pin Settings

For pins that are 5V tolerant, the output can be defined as open drain to allow a 5V output using a pull-up resistor.

Restore Default Settings

This button restores the parameters to the following default settings:

Break on Reset	*not enabled*
Trace on Reset (Foreground)	*not enabled*
Trace Inside Functions (Foreground)	*not enabled*
Trace on Reset (Background)	*not enabled*
Trace Inside Functions (Background)	*not enabled*
Disable Busy/Ready status on SOUT	*not enabled*
Use PIC format (IEEE 754 is default)	*not enabled*
Idle Mode Power Saving Enabled	*enabled*
Sleep Mode Power Saving Enabled	*not enabled*
External Input	*D8, rising edge*
Interface Mode	*SEL pin selects interface (default)*
I²C address	*C8*
Auto-Start Mode>Disable Debug	*not enabled*
Auto-Start Mode>Call Function	*not enabled*
3.3V / 5V (Open Drain) Pin Settings	*all set to 3.3V*

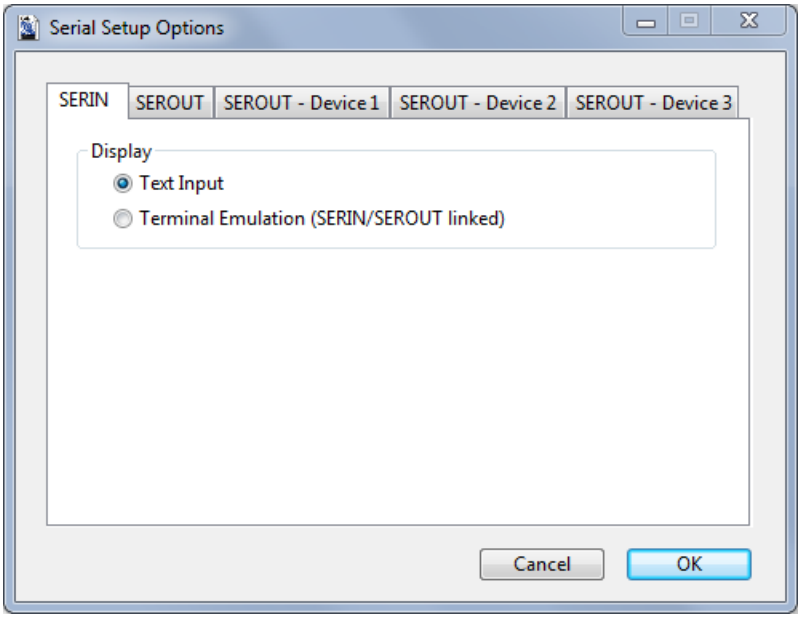
Reference Guide: SERIN and SEROUT Support

The uM-FPU64 IDE uses the `SERIN` and `SEROUT` pins for communication with the debug monitor.
The uM-FPU64 V402 firmware provides the IDE with the capability to debug a project that uses the `SERIN` and `SEROUT` pins, and to receive serial data from multiple serial devices. If the debug monitor is enabled, the FPU communicates with the IDE to get data for the SERIN instruction, and sends data to the IDE from the `SEROUT` instruction. The `SEROUT` instruction supports three extra devices that can be used for sending data to the IDE. If the debug monitor is not enabled, output from the additional `SEROUT` devices is suppressed.

Note: To use the IDE support for the `SERIN` and `SEROUT` instructions, the debug monitor on the FPU must be active. All `SEROUT,SET_BAUD` instructions that disable the debug monitor must be commented out while debugging.

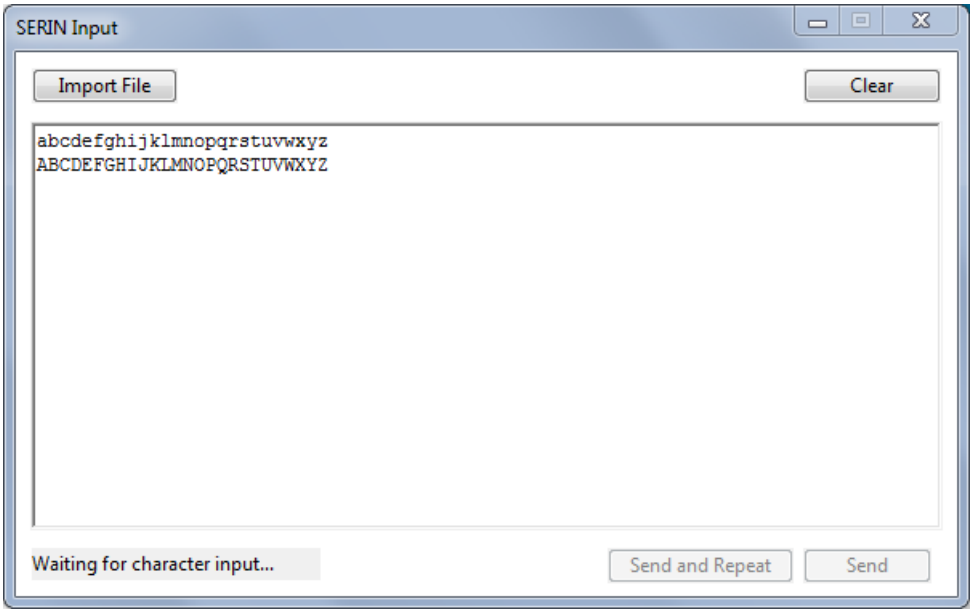
SERIN Window Setup Options

The SERIN window is configured using the *Window>Show Serial Window>Setup Options* menu item. It can be configured for *Text Input* or *Terminal Emulation* mode. In *Terminal Emulation* mode, serial input and output are both handled by the SEROUT window.



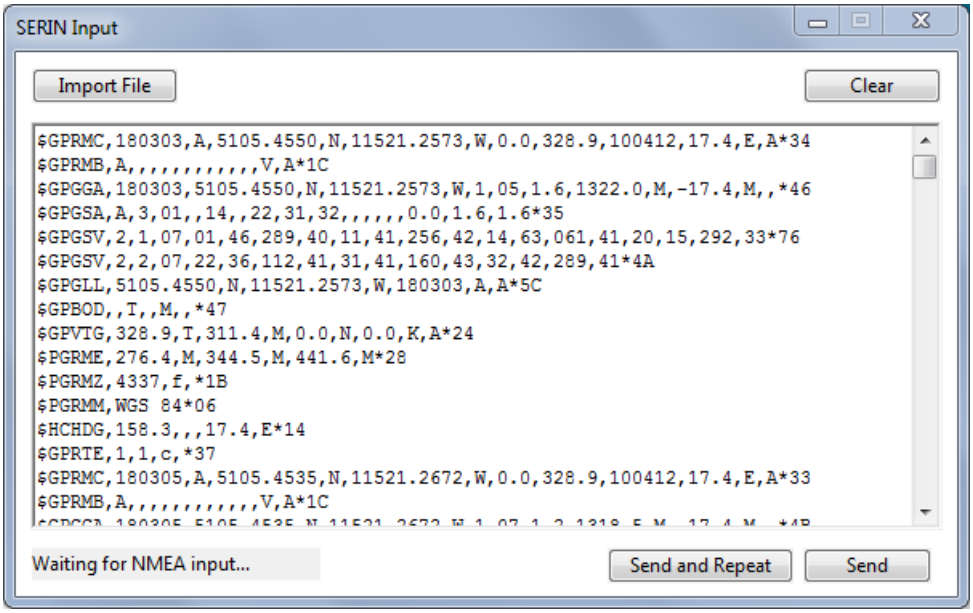
Text Input - Character Mode

When the `SERIN,ENABLE_CHAR` instruction is executed the IDE enters character mode. When a `SERIN,READ_CHAR` instruction is executed, the IDE waits for the user to send the next character. The characters to send can be entered manually in the SERIN window or imported from a text file. In *Text Input* mode, the text is not actually sent to the FPU until you select a character or group of characters, and press one of the send buttons. The *Send* button sends the single character at the start of a selection. The *Send and Repeat* button sends each of the selected characters, in sequence, one at a time, as each `SERIN,READ_CHAR` instruction is executed. The user is not prompted for additional input until the selection has been completely sent. The repeat action can be stopped by making another selection.



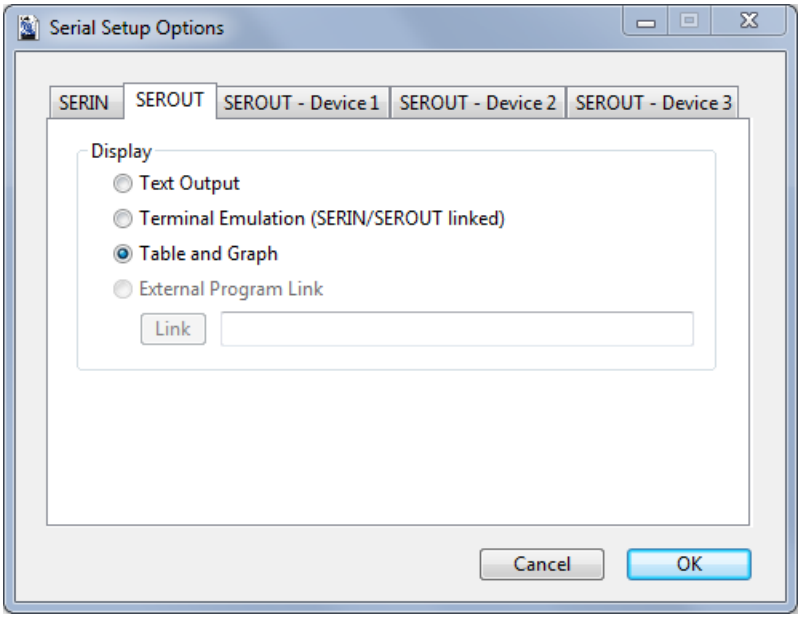
Text Input - NMEA Mode

When the `SERIN,ENABLE_NMEA` instruction is executed the IDE enters NMEA mode. When a `SERIN,READ_NMEA` instruction is executed, the IDE waits for the user to send the next NMEA sentence. The sentences to send could be entered manually in the SERIN window, but they are normally imported from a text file. The sentences are not actually sent to the FPU until you select a sentence or group of sentences, and press one of the send buttons. The *Send* button sends the single sentence at the start of a selection. The *Send and Repeat* button sends each of the selected sentences, in sequence, one at a time, as each `SERIN,READ_NMEA` instruction is executed. The user is not prompted for additional input until the selection has been completely sent. The repeat action can be stopped by making another selection. Only complete sentences are sent to the FPU. If only part of a sentence is selected, the complete sentence will be sent.



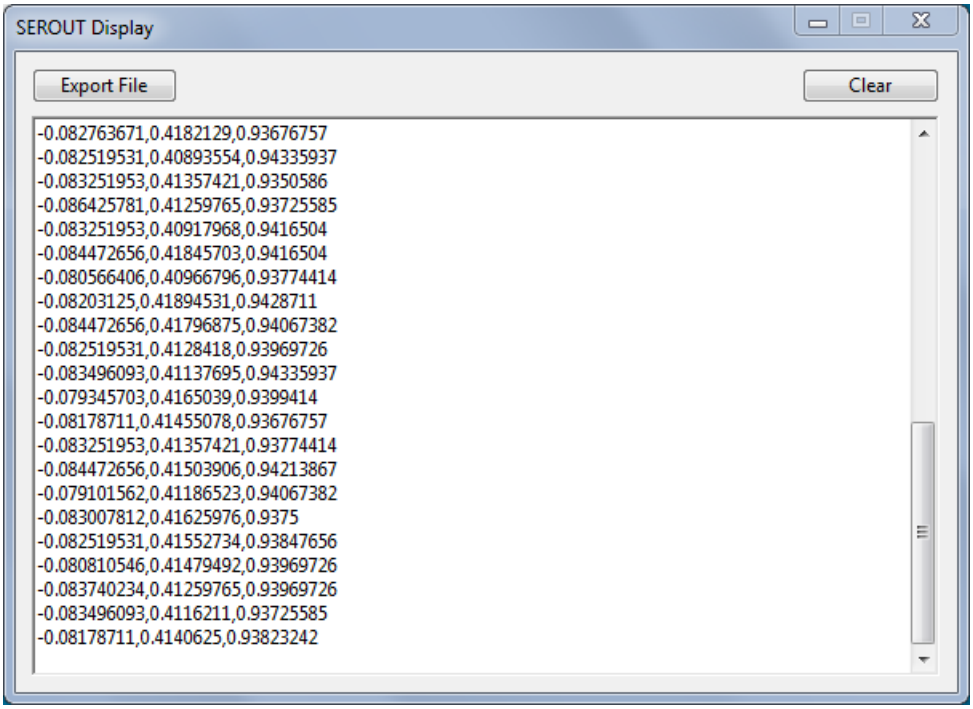
SEROUT Window Setup Options

The SEROUT window is configured using the *Window>Show Serial Window>Setup Options* menu item. It can be configured for *Text Output*, *Terminal Emulation*, or *Table and Graph* mode.



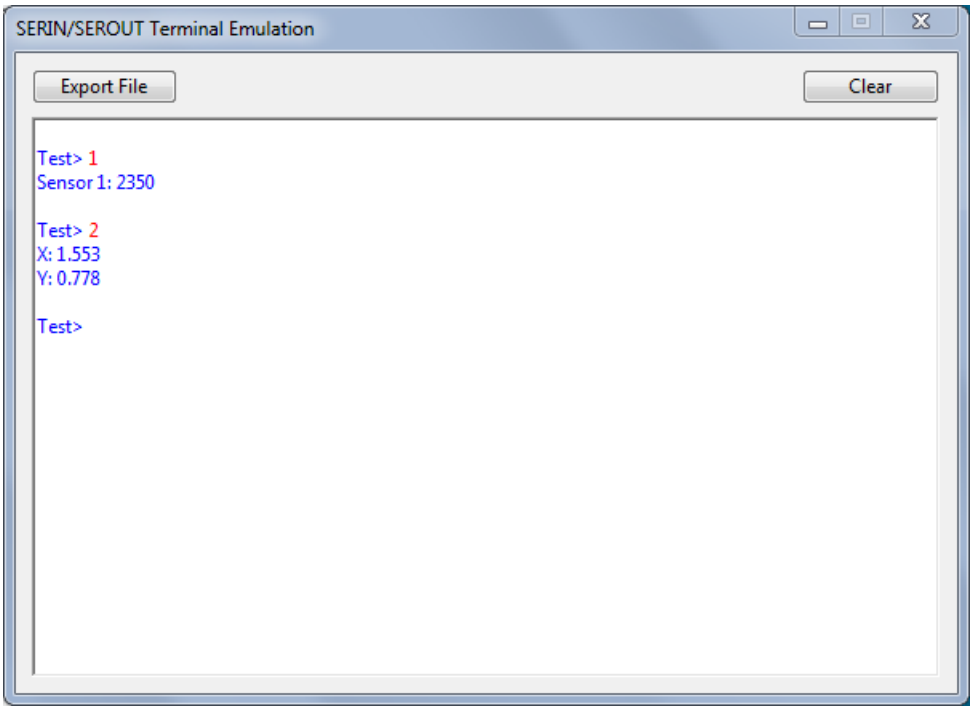
SEROUT Window - Text Output Mode

In *Text Output* mode, data sent by the `SEROUT` instruction is displayed in a text window, in black, with no additional formatting. The text output can be exported to a text file.



SEROUT Window - Terminal Emulation Mode

In *Terminal Emulation* mode, serial input and serial output are both handled by the SEROUT window. Data sent by the `SEROUT` instruction is shown in blue, with no additional formatting. Characters typed by the user are shown in red. They are not displayed until the `SERIN` instruction requests data. A typeahead buffer is provided.

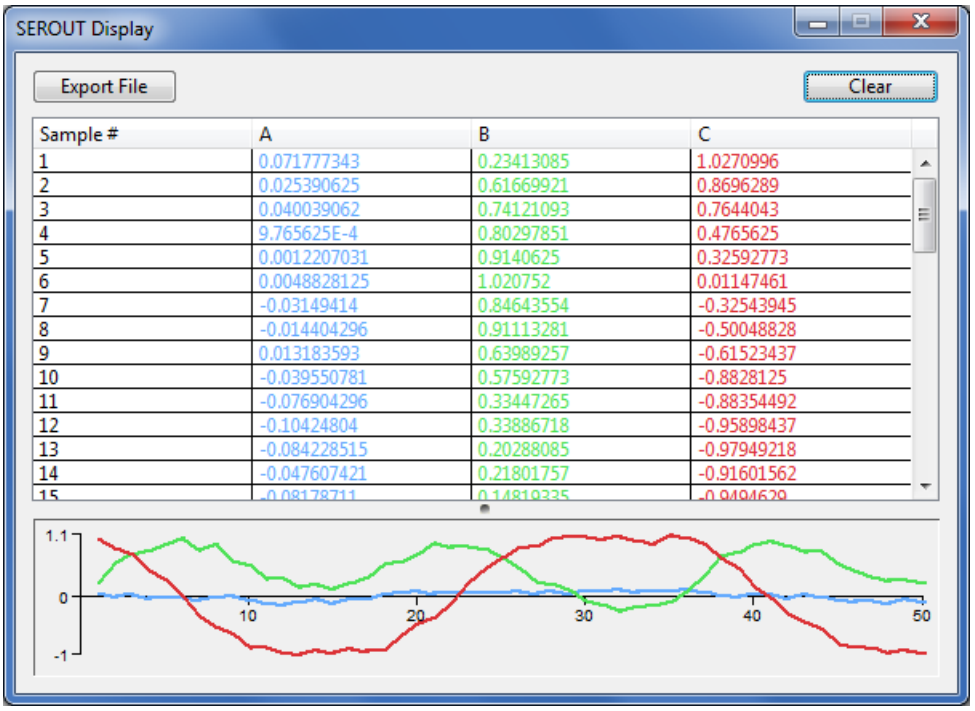


SEROUT Window - Table and Graph Mode

In *Table and Graph* mode, data sent by the `SEROUT` instruction is displayed in a table and graph. The data in each column in displayed in a different color, and each column is graphed using a line of the same color. The X and Y scales for the graph are automatically calculated to display the entire data set.

The data received from the `SEROUT` instruction must be comma separated numbers terminated with a carriage return. The new `SEROUT,WRITE_FLOAT`, `SEROUT,WRITE_LONG`, `SEROUT,WRITE_COMMA`, and `SEROUT,WRITE_CR` instructions make it easy to create comma separated values.

The values in the table can be exported to a comma separated value (CSV) file.



SEROUT Window Device 1, Device 2, Device 3 Setup Options

The SEROUT - Device1, SEROUT - Device 2, and SEROUT - Device 3 windows are configured using the *Window>Show Serial Window>Setup Options* menu item. They can be configured for *Text Output* or *Table and Graph* mode. The capabilities of these modes are the same as described for the SEROUT window, with the exception of *Terminal Emulator* mode, which is only available for the SEROUT window.