Micromega Corporation

uM-FPU64 IDE
Integrated Development Environment

User Manual

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. *r401*). 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*.

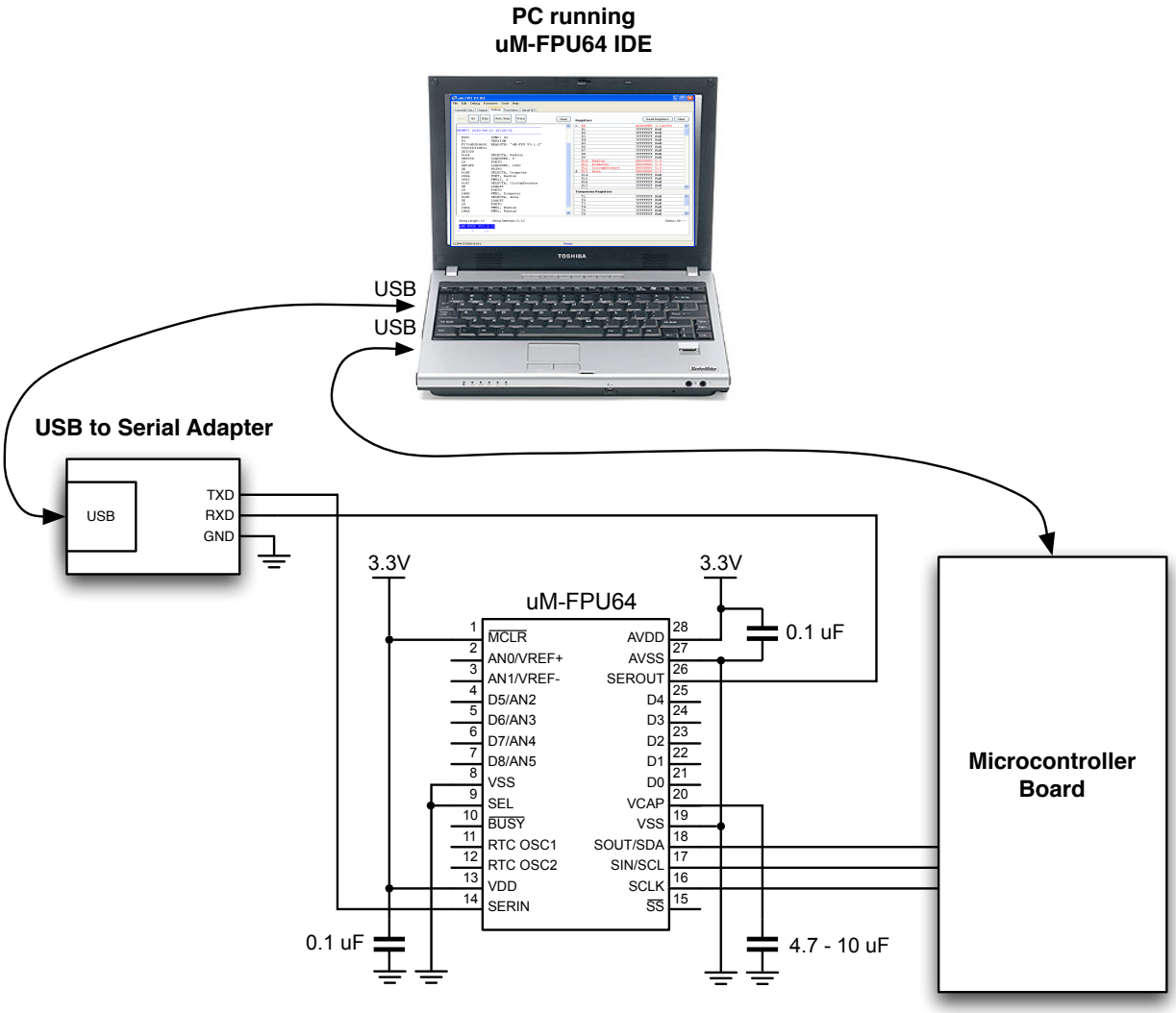
Connecting to the uM-FPU64 chip

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).

Examples of suitable USB to Serial adapters include:

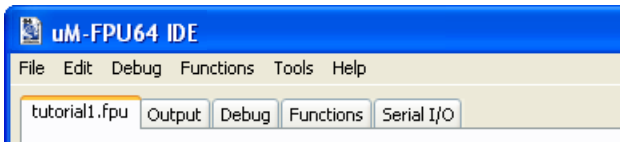
Sparkfun	FTDI Basic Breakout - 3.3V	http://www.sparkfun.com/
Parallax	USB2SER Development Tool	http://www.parallax.com/

Connection Diagram



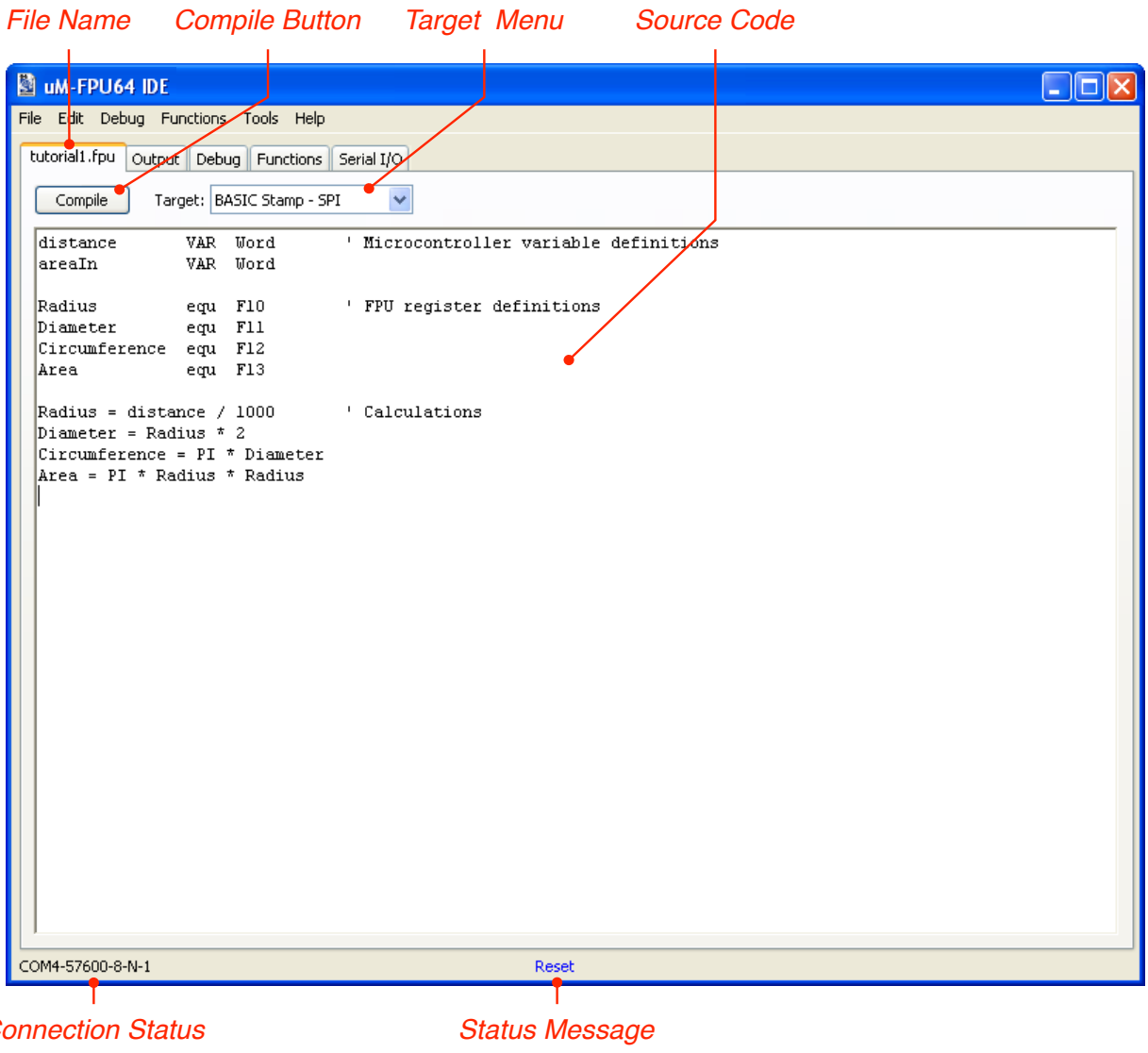
Overview of uM-FPU64 IDE User Interface

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*.

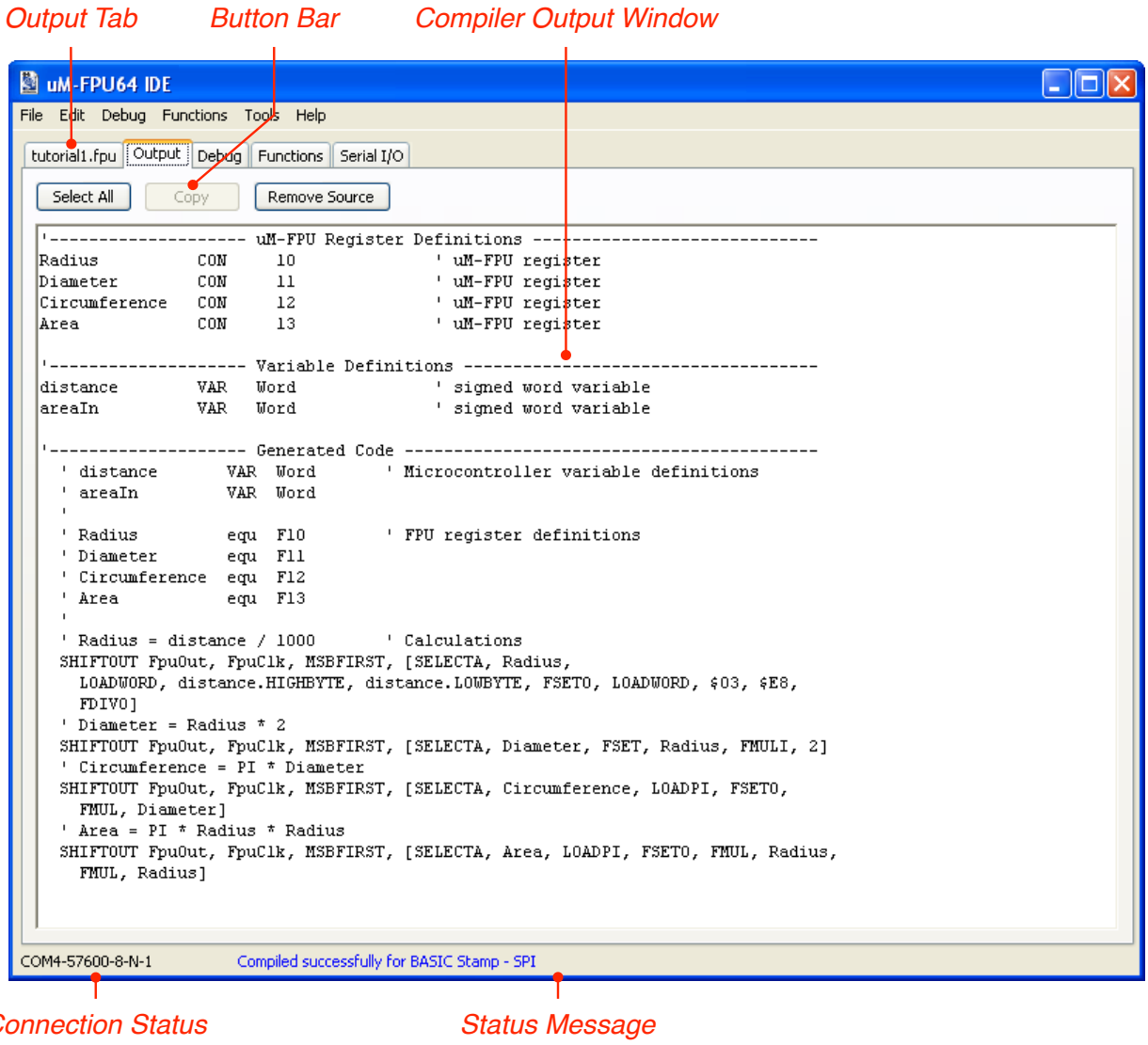


The ***Source Window*** is used to edit the source code and compile the source code. Pressing the ***Compile*** button

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

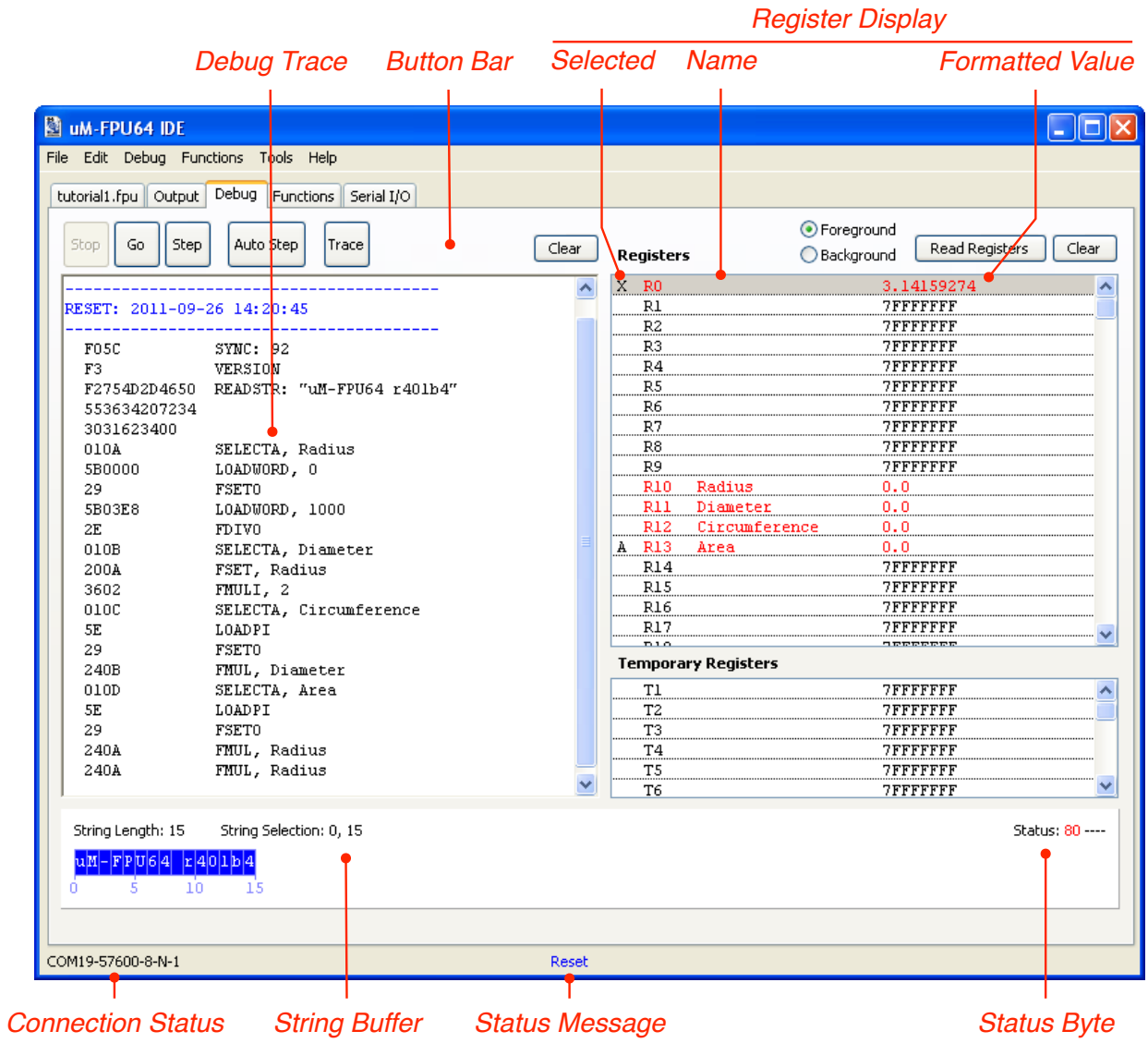
The ***Output Window*** is automatically displayed if the compile is successful. The status message will show that the compile was successful. All normal status messages are displayed in blue.



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.

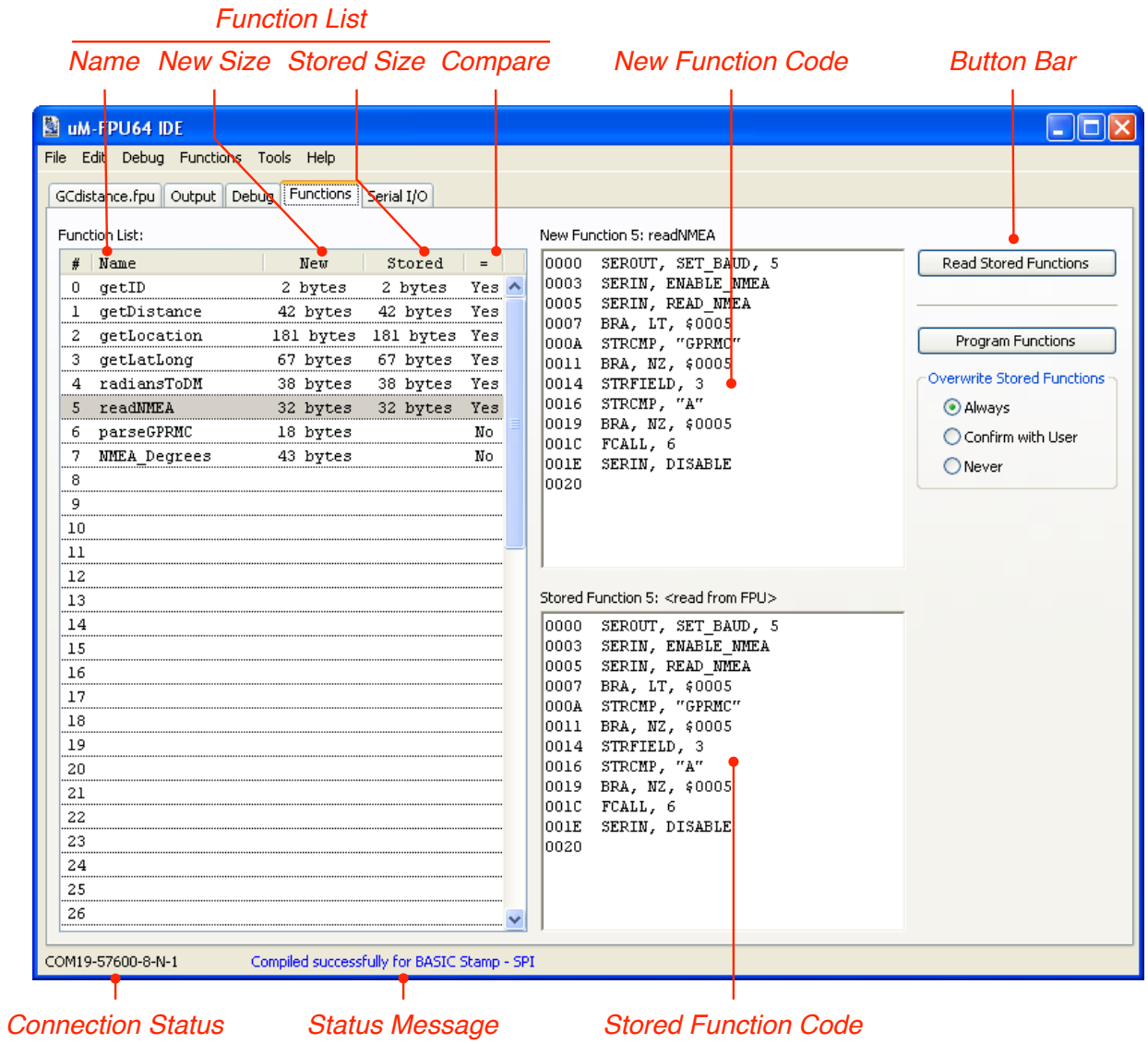


The ***Debug Trace*** displays messages and instruction traces. The Reset message includes a time stamp, is displayed whenever a hardware or software reset occurs. Instruction tracing will only occur if tracing is enabled. This can be enabled at Reset by setting the ***Trace on Reset*** option in the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

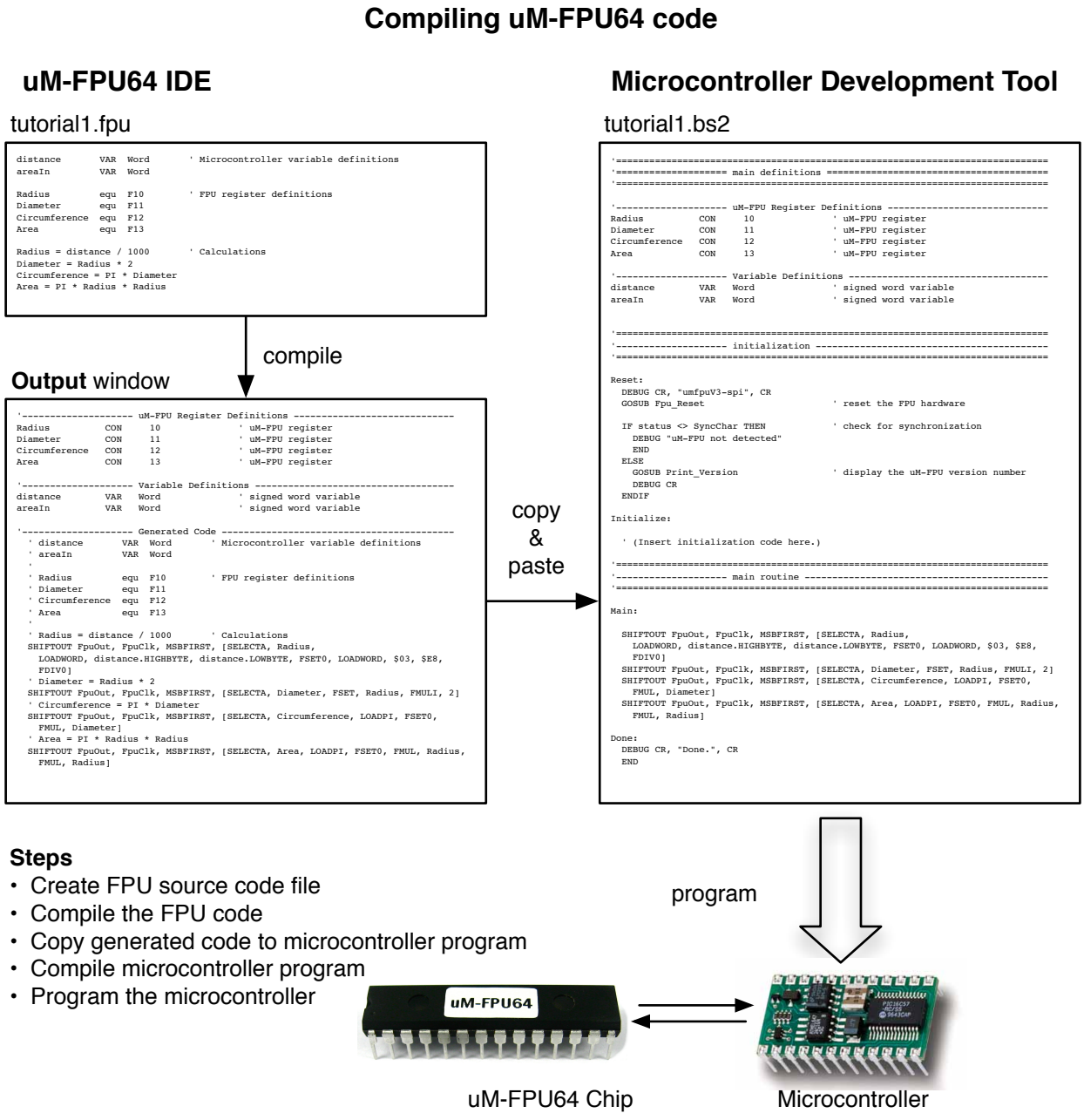
Serial I/O Window

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

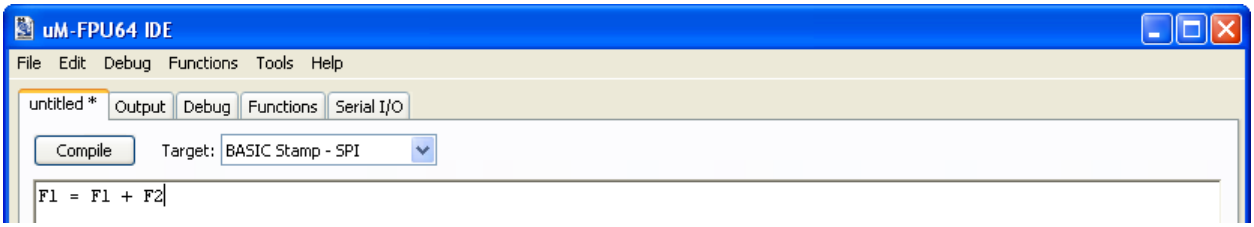
The uM-FPU64 IDE has predefined names for the registers in the FPU.

`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
```

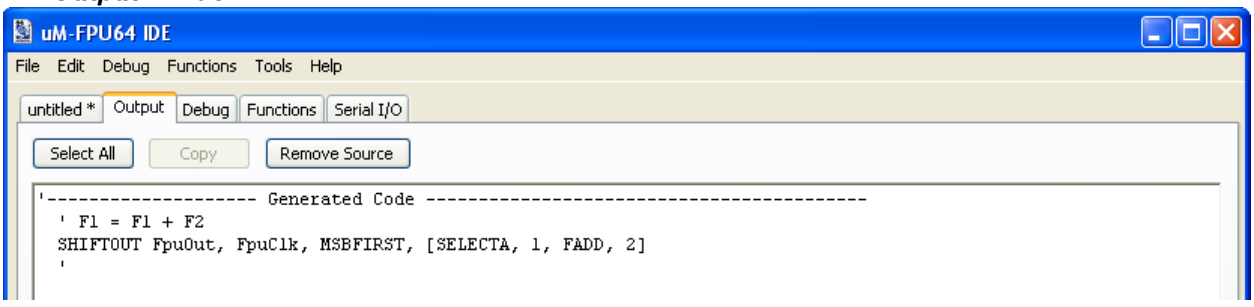
The ***Source Window*** should look as follows:



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.

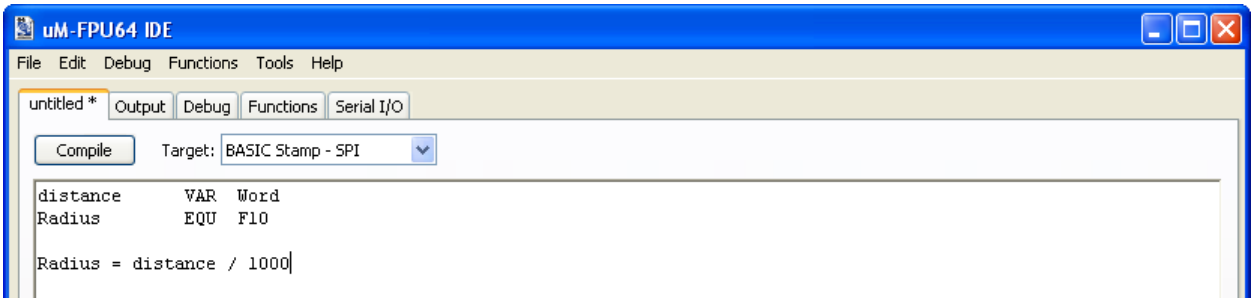
Calculating Radius

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
```

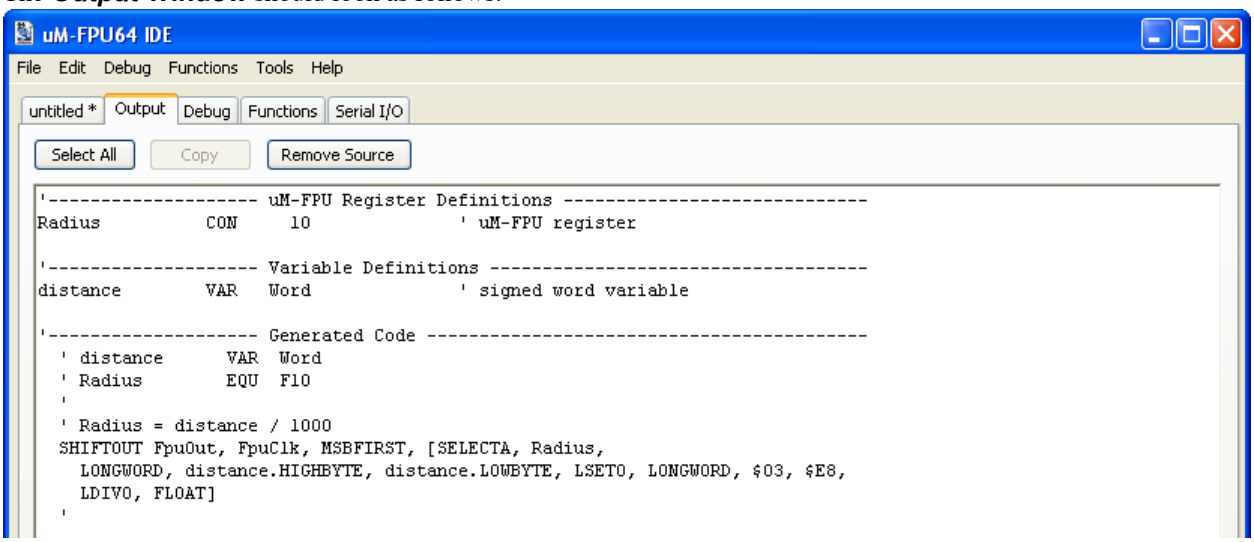
The ***Source window*** should look as follows:



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
```

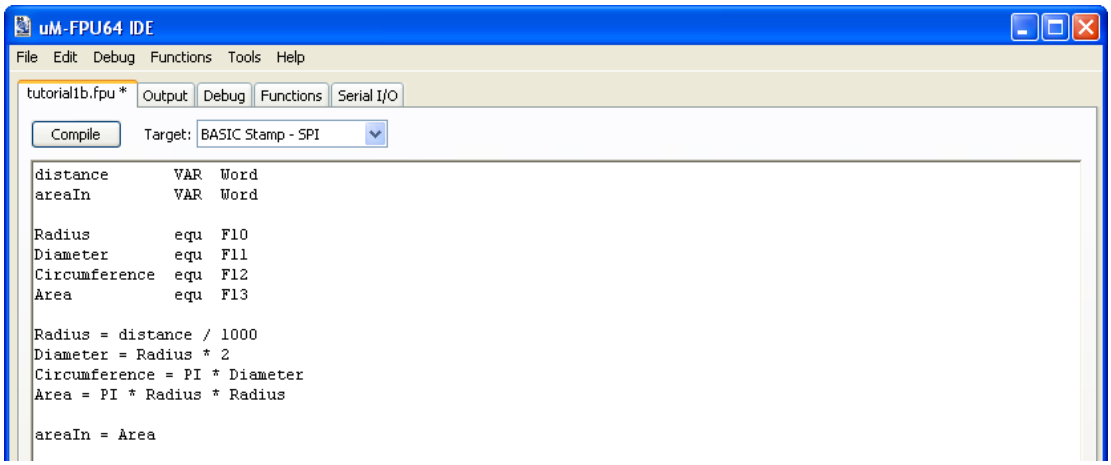
Finally, we'll read the `Area` value back to the microcontroller as a 16-bit integer and print the result. To do this we first add the following definition for the microcontroller variable:

```
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.

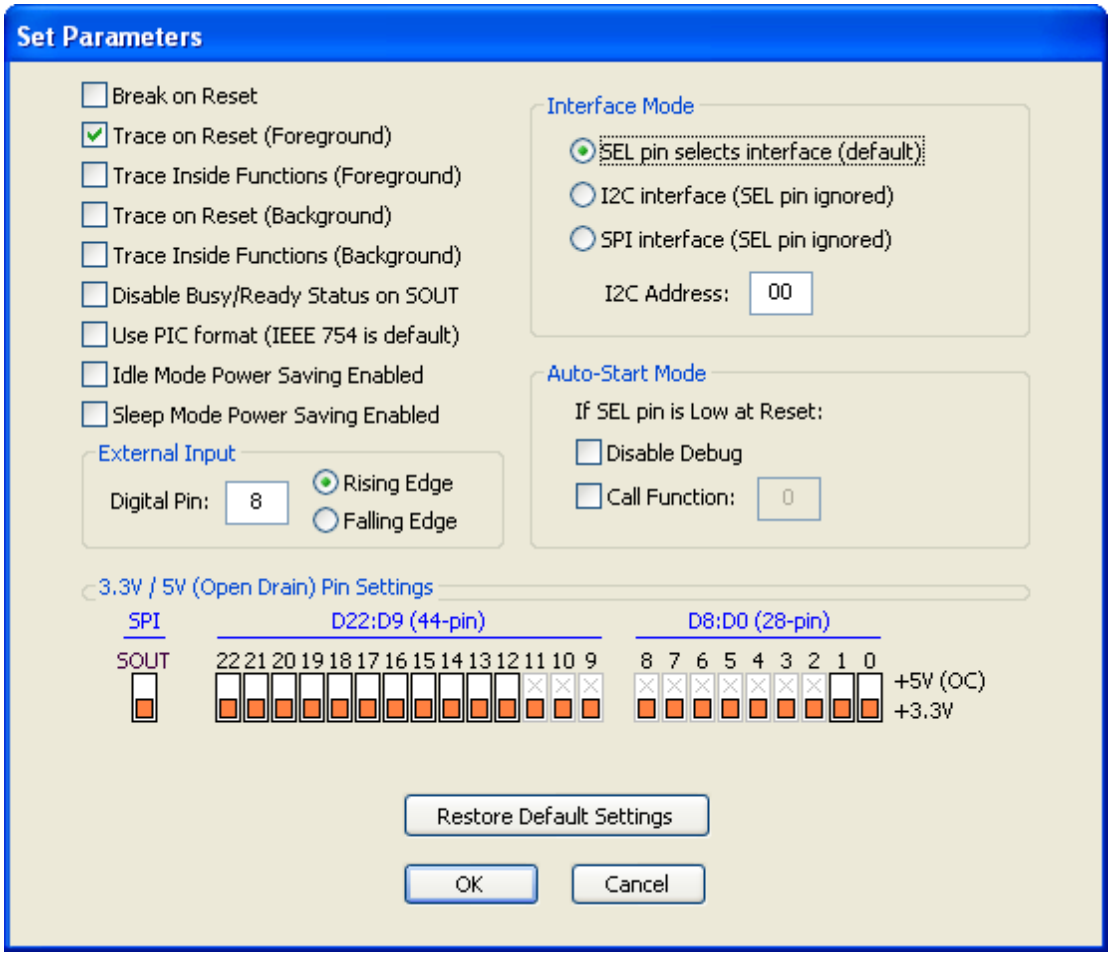
Making the Connection

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

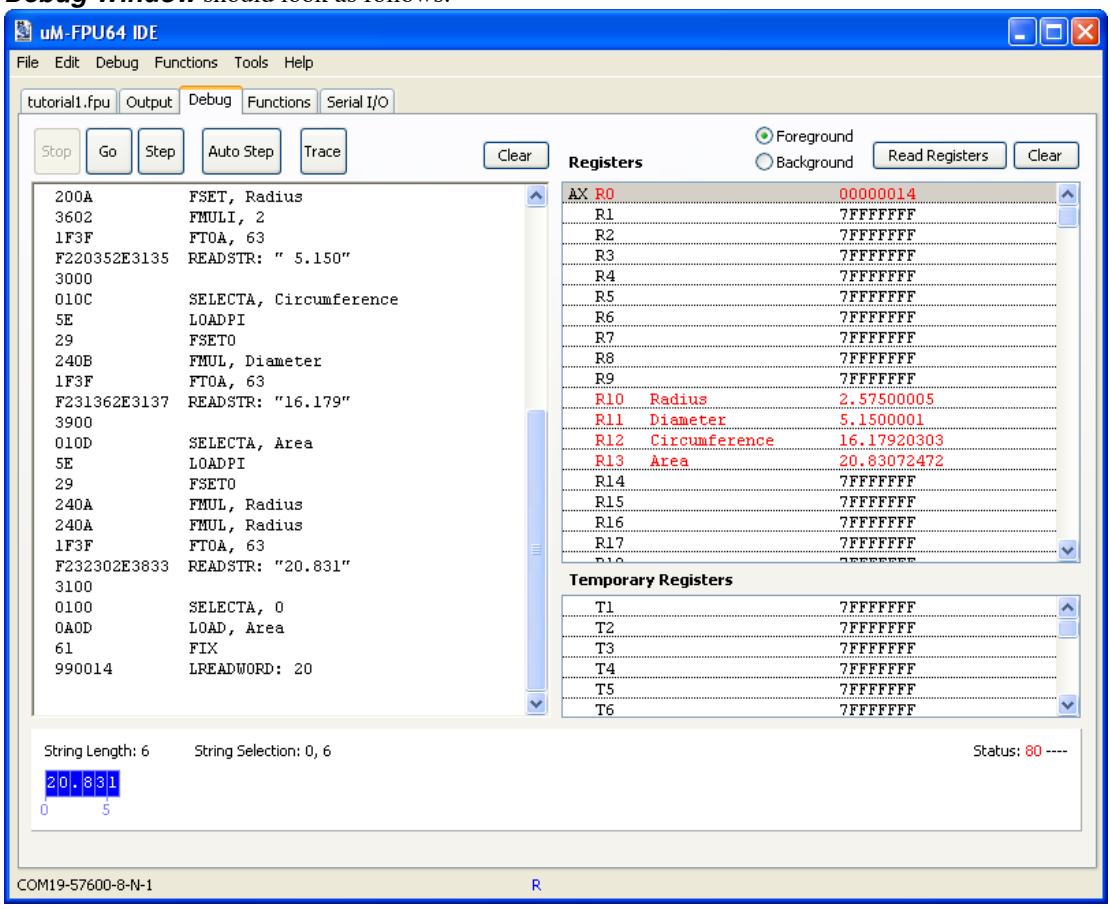
The ***Debug Window*** of the IDE can display a trace of all instructions as they are executed. By default, tracing is disabled. It can be enabled at Reset by setting the ***Trace on Reset (Foreground)*** option in the ***Functions>Set Parameters...*** dialog, or it can be turned on or off at any time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

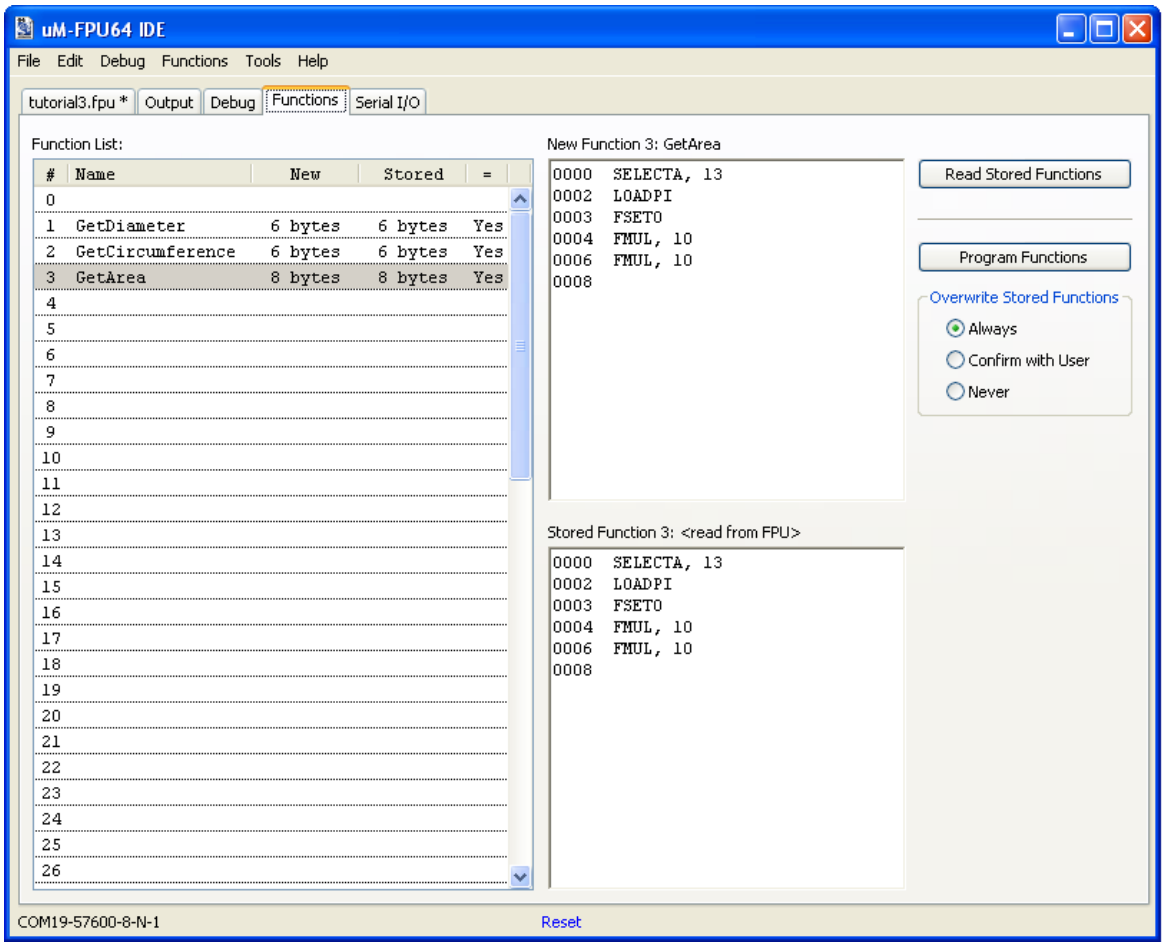
Save the file as *tutorial3.fp4.*

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:

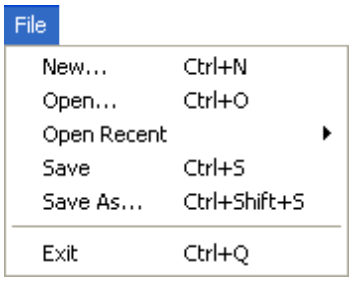
```
uM-FPU64 r401b4

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.

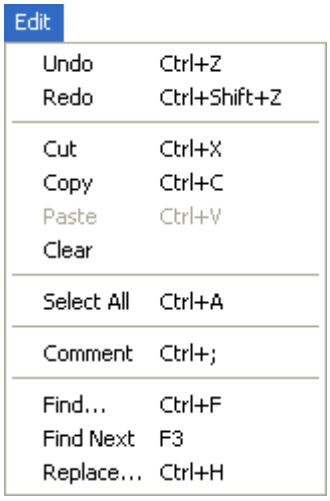
Open Recent menu item provides a sub-menu that lists up to ten source files that were recently saved. Selecting a source file from the sub-menu will open the file. If a previous source file is open and has been changed since the last time it was saved, you will first be prompted to save the previous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

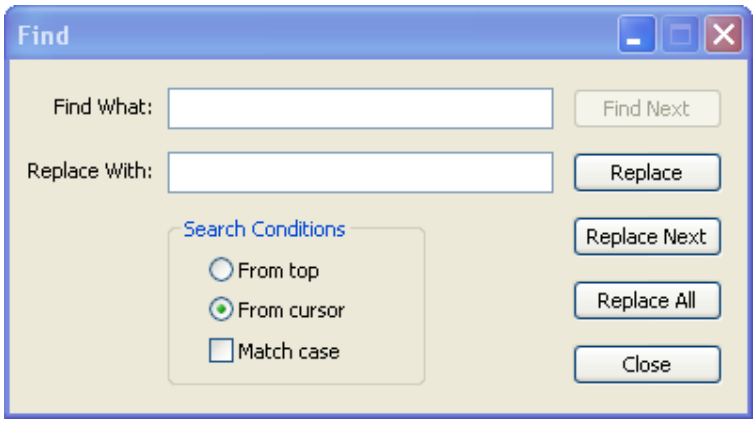
Clear menu item deletes the selected text from the ***Source Window***.

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.



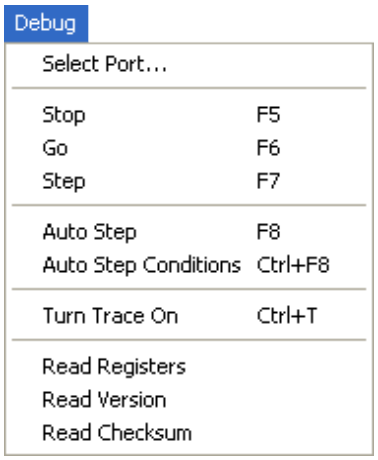
The **Find** dialog is a moveable dialog and can be placed alongside the ***Source Window*** and left open when multiple find and replace operations are done. The **Find What** field specified the string to search for, and the **Replace With** field specifies the string to replace it with. If the **From 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.

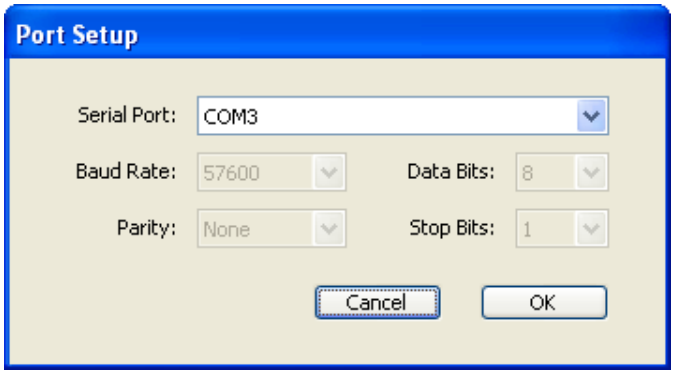
Find Next menu item finds the next match based on the current search conditions in the **Find** dialog.

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.

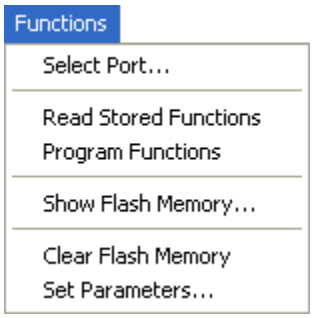
Auto Step menu item continues execution in auto step mode. See the section entitled *Reference Guide: Auto Step and Conditional Breakpoints* for more details.

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

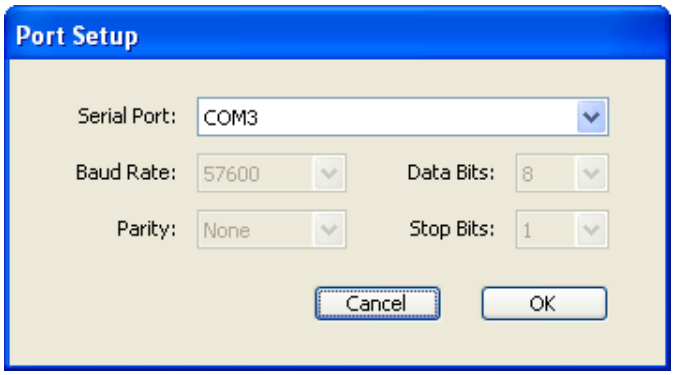
Read Version menu item will display the version of the FPU in the ***Debug Trace***.

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

Functions Menu



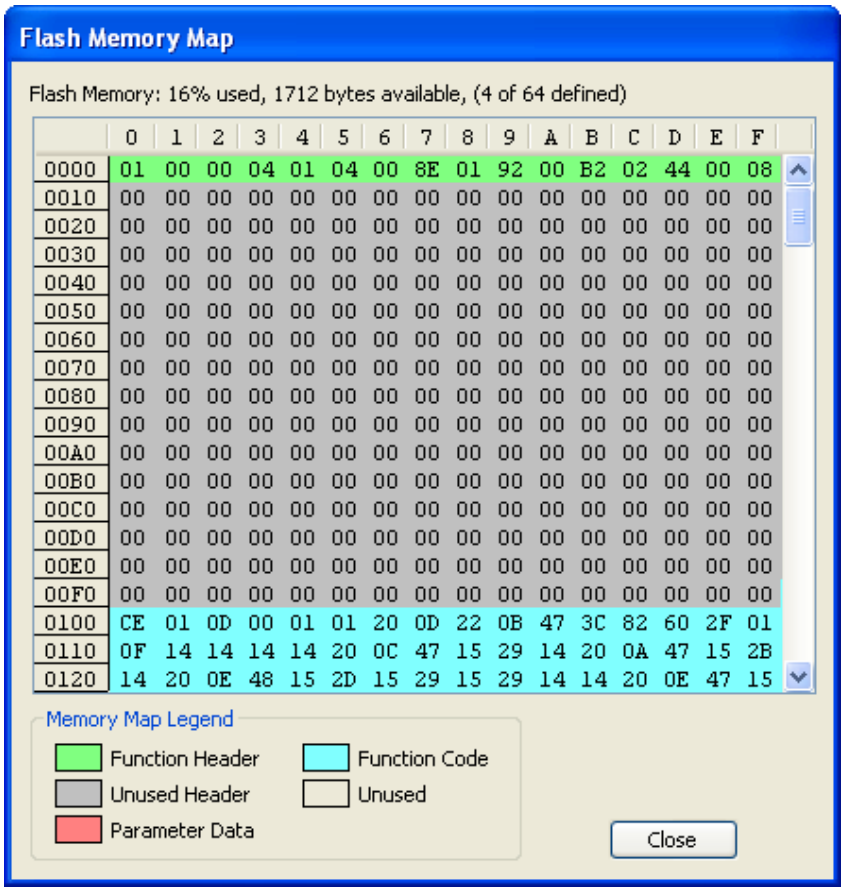
Select Port... menu item is used to display the **Port Setup** dialog which is used to select the serial communications port.



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.

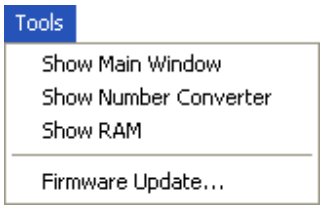
Show Flash Memory... menu item displays a memory map showing the usage of the Flash memory reserved for user-defined functions on the uM-FPU64 chip. A status line at the top shows the percent of memory used and the number of bytes available.



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

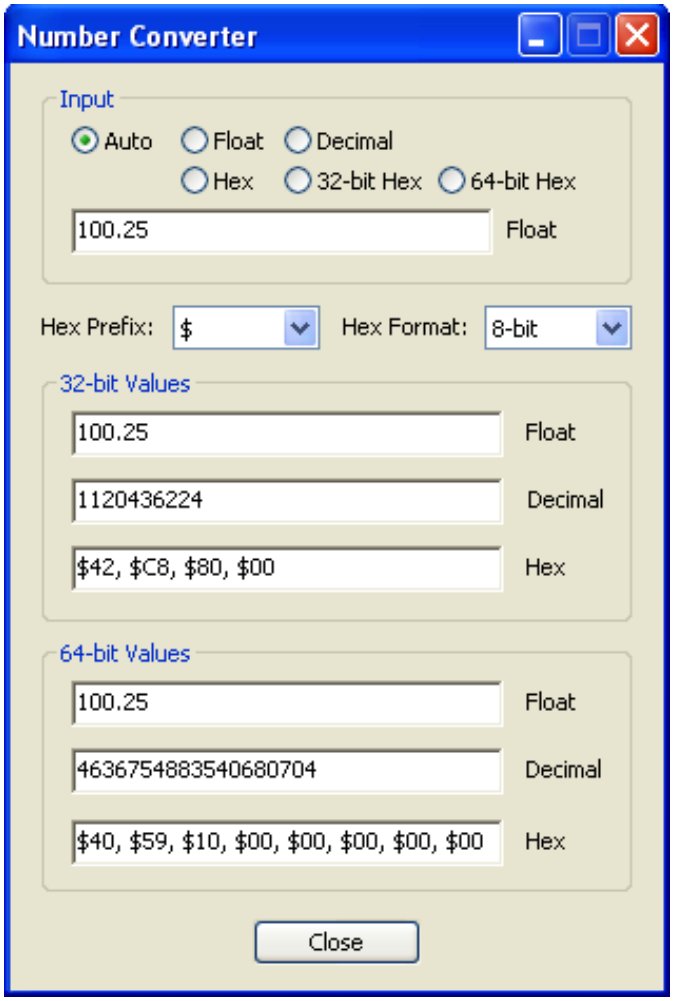


Show Main Window menu item is used to bring the main IDE window to the front.

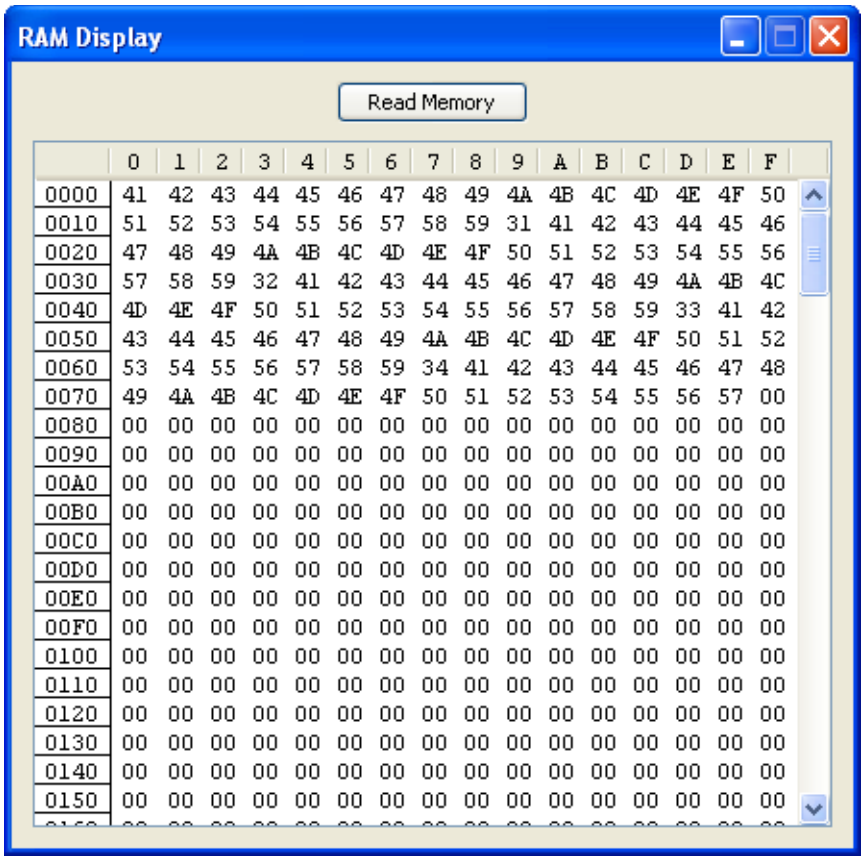
Show Number Converter menu item is used to bring the **Number Converter** window to the front. The number converter provides a quick way to convert numbers between various 32-bit and 64-bit formats. Floating point, decimal and hexadecimal numbers are supported. The **Auto**, **Float**, **Decimal**, and **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.



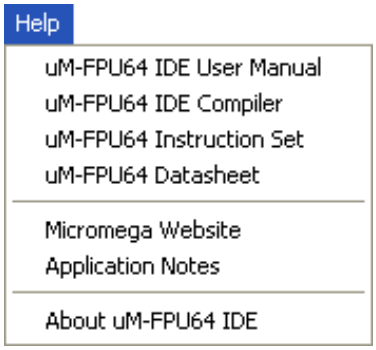
Show RAM menu item is used to bring the **RAM Display** window to the front. This window is used to view the contents of RAM.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50
0010	51	52	53	54	55	56	57	58	59	31	41	42	43	44	45	46
0020	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53	54	55	56
0030	57	58	59	32	41	42	43	44	45	46	47	48	49	4A	4B	4C
0040	4D	4E	4F	50	51	52	53	54	55	56	57	58	59	33	41	42
0050	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52
0060	53	54	55	56	57	58	59	34	41	42	43	44	45	46	47	48
0070	49	4A	4B	4C	4D	4E	4F	50	51	52	53	54	55	56	57	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	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0110	00	00	00	00	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
0130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

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

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

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.

Order of Evaluation

Math equations are evaluated by the IDE from left to right with no operator precedence.

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

results in `F1` being set to the value of `F2` added to `F3`, then multiplied by `F4`. Parentheses must be used to change the order of operations.

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

results 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.

```
F1 = F2 * 5 / 2
```

results 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 in the above equation, the equation would be calculated as `F2` multiplied by `33.8`, which is incorrect. The code generator often adds one level of parenthesis, so parentheses in math equations should only be nested up to seven levels deep, including the parentheses used for functions.

Comments

Comments can be added to any line of source code. Comments are preceded by an apostrophe, semi-colon or double slash characters. All text after the comment prefix to the end of line is considered a comment.

```
' all text after an apostrophe to the end of line is a comment
; all text after a semi-colon to the end of line is a comment
// all text after a double slash to the end of line is a comment
```

Symbol Names

Symbol names must begin with an alphabetic character, followed by any number of alphanumeric characters or the underscore character. Symbol names can be defined for FPU registers, constants, microcontroller variables, and functions. They are not case-sensitive. Here are some examples:

```
getDistance
latitude1
NMEA_Degrees
```

Register Data Types

The 32-bit and 64-bit FPU registers can be defined as `Float`, `Long` or `Unsigned` data types.

`Float`	32-bit or 64-bit IEEE 754 format
`Long`	32-bit or 64-bit signed integer
`Unsigned`	32-bit or 64-bit unsigned integer

Pre-defined Register Names

The uM-FPU64 chip has 256 FPU registers. Registers 0 to 127 are 32-bit registers, and register 128 to 255 are 64-bit registers. The following register names are pre-defined:

`F0, F1, F2, ... F255`	specifies that register 0 to 255 contains a Float data type
`L0, L1, L2, ... L255`	specifies that register 0 to 255 contains a Long data type
`U0, U1, U2, ... U255`	specifies that register 0 to 255 contains an Unsigned data type

User-defined Register Names

User-defined names can be assigned to registers with the **`EQU`** operator. The user-defined register name on the left of the **`EQU`** operator is set to the value of the pre-defined register name on the right. For example:

```
reg0                EQU  F0
tmp1                EQU  F1
Y                   EQU  F10
X                   EQU  F11
Radius              EQU  F12
shoulderPulseRate   EQU  L13
```

Decimal Constants

Decimal constants are represented as a sequence of decimal digits (without commas, spaces, or periods), with optional + or - prefix.

```
  120      -53     100000      +207
```

Hexadecimal Constants

Hexadecimal constants must have a **`0x`** or **`$`** prefix and are represented as a sequence of hexadecimal digits (without commas, spaces, or periods). The hexadecimal digits and prefix can be upper or lower case.

```
  $55     0xFF      $FFFF      0x13
```

Floating Point Constants

Floating point constants consist of an optional **`+`** or **`-`** prefix, decimal integer, decimal point, decimal fraction, **`e`** or **`E`**, and a signed integer exponent. Only the decimal integer is required, the other fields are optional. If the **`e`** or **`E`** is used an integer exponent must follow.

```
  1.0      -53        1E6    -1.5e-3
```

Pre-defined Constants

`PI`	constant value for pi (32-bit: 3.1415926, 64-bit: 3.141592653589793)
`E`	constant value for e (32-bit: 2.7182818, 64-bit: 2.718281828459045)

User-defined Constants

User-defined constants can be defined with the **`CON`** or **`EQU`** operator. The user-defined constant on the left of the

CON or **EQU** operator is set to the value of the constant expression on the right. The compiler simplifies constant expressions to a single constant value. For example:

e.g.

```
Length  CON  4.75
Pi2     CON  PI / 2
```

or

```
Length  EQU  4.75
Pi2     EQU  PI / 2
```

String Constants

A string constant is enclosed in double quote characters. Special characters can be entered using a backslash followed by two hexadecimal digits. The backslash and double quote characters can be entered by preceding them with a backslash.

String Constant	Actual String
`"sample"`	`sample`
`"string2\0D\0A"`	`string2<`*carriage return*`><`*linefeed*`>`
`"5\\3"`	`5\3`
`"this \"one\" "`	`this "one"`

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

```
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 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, then add 32.

```
degreesC     EQU    BYTE
degreesF     EQU    F10

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

Special syntax for PICAXE

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

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

Math Operators

The following math operators can be used for Float, Long and Unsigned data types.

+	Plus
-	Minus
*	Multiply
/	Divide

```
x = -y * z / 2
```

Math Functions

The following math functions are provided. Each of the functions uses an FPU instruction of the same name (ABS, MOD, MIN and MAX use the FABS, FMOD, FMIN, FMAX instructions for floating point data types, and the LABS, LDIV (remainder), LMIN, LMAX instructions for Long or Unsigned data types). More detailed information on the functions can be obtained by referring to the corresponding FPU instruction in the *uM-FPU64 Instruction Set* document.

Function	Arguments	Return	Description
SQRT(*arg1*)	Float	Float	square root of *arg1*.
LOG(*arg1*)	Float	Float	logarithm (base e) of *arg1*.
LOG10(*arg1*)	Float	Float	logarithm (base 10) of *arg1*.
EXP(*arg1*)	Float	Float	e to the power of *arg1*.
EXP10(*arg1*)	Float	Float	10 to the power of *arg1*.
SIN(*arg1*)	Float	Float	sine of the angle *arg1* (in radians).
COS(*arg1*)	Float	Float	cosine of the angle *arg1* (in radians).
TAN(*arg1*)	Float	Float	tangent of the angle *arg1* (in radians).
ASIN(*arg1*)	Float	Float	inverse sine of the value *arg1*.
ACOS(*arg1*)	Float	Float	inverse cosine of the value *arg1*.
ATAN(*arg1*)	Float	Float	inverse tangent of the value *arg1*.
ATAN2(*arg1, arg2*)	Float	Float	inverse tangent of the value *arg1* divided by *arg2*.
DEGREES(*arg1*)	Float	Float	angle *arg1* converted from radians to degrees.
RADIANS(*arg1*)	Float	Float	angle *arg1* converted from degrees to radians.
FLOOR(*arg1*)	Float	Float	floor of *arg1*.
CEIL(*arg1*)	Float	Float	ceiling of *arg1*.
ROUND(*arg1*)	Float	Float	*arg1* rounded to the nearest integer.
POWER(*arg1, arg2*)	Float	Float	*arg1* raised to the power of *arg2*.
ROOT(*arg1, arg2*)	Float	Float	*arg2* root of *arg1*.
FRAC(*arg1*)	Float	Float	fractional part of *arg1*.
INV(*arg1*)	Float	Float	the inverse of *arg1*.
FLOAT(*arg1*)	Long	Float	converts *arg1* from long to float.
FIX(*arg1*)	Float	Long	converts *arg1* from float to long.
FIXR(*arg1*)	Float	Long	rounds *arg1* then converts from float to long.
ABS(*arg1*)	Float/Long	Float/Long	absolute value of arg1.
MOD(*arg1, arg2*)	Float/Long	Float/Long	the remainder of *arg1* divided by *arg2*.
MIN(*arg1, arg2*)	Float/Long	Float/Long	the minimum of *arg1* and *arg2*.
MAX(*arg1, arg2*)	Float/Long	Float/Long	the maximum of *arg1* and *arg2*.

```
theta = sin(angle)
fcube = power(f, 3)
result = cos(PI/2 + sin(theta))
```

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.

```
#FUNCTION 1 GetDiameter          function definition
```

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 **@** character followed by the name of a previously defined function.

```
@1                               call function 1
@AddValue                        call function AddValue
```

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

```
Value1 EQU BYTE                  microprocessor variable definitions
Value2 EQU BYTE

X      EQU F1                    register definitions
Y      EQU F2
Z      EQU F3

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

X = Value1
Y = Value2
@Hypotenuse                      call function Hypotenuse
```

Function calls can be nested up to 16 levels deep.

Function Prototypes

To ensure that the function being called is already defined, function prototypes can be included at the start of the program. 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, which assigns a symbol name to a function number. You can assign the function number explicitly, or use the **%** character to assign the next unused function number.

```
GetDiameter         func   1     GetDiameter is function 1
GetCircumference    func   %     GetCircumference is function 2
GetArea             func   %     GetArea is function 3
```

Global Symbols vs Local Symbols

All symbols defined at the top of the source file, outside of any function, are global symbols, and can be used by any source code that follows. Symbols that are defined inside a function, are local symbols, and can only be used within that function.

```
tmp1   equ    F1                global symbol definition

#function sample1
tmp2   equ    F2                local symbol definition

    SELECTA, tmp1                both tmp1 and tmp2 are defined inside the function
```

```
    FSET, tmp2
#end
```

only tmp1 is defined outside the function

Assembler Code

The IDE compiler converts regular math equations in the source code into the required uM-FPU64 instructions for performing the calculation. Some capabilities of the uM-FPU64 chip are not accessible using the compiler, or in some cases it may be possible to write more optimized code using assembler. Assembler code can be entered by enclosing it with the **#ASM** and **#ENDASM** directives. See the next section entitled *Reference Guide: Assembler* for more details on assembler code.

```
#ASM                          start of assembler
  SELECTA, 1
  LOADPI                      assembler code
  FSET0
  FDIVI, 2
#ENDASM                       end of assembler
```

Wait Code

The uM-FPU64 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.

Reference Guide: Assembler

Assembler code can be entered by enclosing it with the **#ASM** and **#ENDASM** directives. 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
```

single line of assembler

or

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

multiple lines of assembler

Assembler Instructions

All assembler instructions start with an opcode followed by any required arguments (if any) separated by commas. Opcode names and symbol names may be entered in uppercase or lowercase, they are not case sensitive. The following table summarizes the syntax for each instruction and the required arguments. Please refer to the *uM-FPU64 Instruction Set* document for a more detailed description of the 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
LCOPYI, 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
FCOPYI, bb, reg
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
SAVEMA, bb, bb
```

```
SAVEMB, bb, bb
SAVEMC, bb, bb
MOP, bb
FFT, bb
WRIND, bb, bb...
RDIND, bb
DWRITE, reg, float64val
DREAD, reg
LBIT, bb, reg
SETIND, bb, bb
ADDIND, reg, bb
COPYIND, reg, reg, reg
LOADIND, reg
SAVEIND, reg
INDA, reg
INDX, reg
FCALL, fnum
EVENT, bb
RET
BRA, _label
BRA, cc, _label
JMP, _label
JMP, cc, _label
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
LSHIFTI, bb
LANDI, bb
LORI, bb
DIGIO, bb
ADCMODE, bb
ADCTRIG
ADCSCALE, bb
ADCLONG, bb
ADCLOAD, bb
ADCWAIT
TIMESET
TIMELONG
TICKLONG
DEVIO, dev, bb…
DELAY, wwww
RTC, bb
SETARGS
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
SETREAD
RESET
```

Where:

`reg`	register number (0-127)
`fnum`	Flash function number (0-63)
bb	8-bit value
bb…	multiple 8-biy values
dev	device
`wwww`	16-bit value
`_label`	address label
`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

The following directives can be used to define byte, word, long and float values.

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

```
POLY, 3
#float -2.8E-6
#float 0.0405
#float -4.0
```

POLY instruction with coefficients -0.0000028, 0.0405, -4.0

The following directives generate code to print to a terminal window (e.g. the built-in terminal window of the target microcontroller IDE). The commands used for output are defined in the target description file.

`#PRINT_FLOAT format`	*print floating point value (if no format specified, 0 is assumed)*
`#PRINT_LONG format`	*print integer value (if no format specified, 0 is assumed)*
`#PRINT_FPUSTRING`	*print FPU string*
`#PRINT_STRING string`	*print string constant*
`#PRINT_NEWLINE`	*print new line (e.g. carriage return, linefeed)*

Symbol Definitions

All symbols that have been defined by the compiler can be used by the assembler code.

```
angle  EQU    F10
```

symbol definition

```
#asm
   SELECTA, angle
#endasm
```

symbol used by assembler instruction

Branch and Return Instructions

Branch instructions are only valid inside a function. There are four types of branch instructions, and a computed `GOTO` instruction.

`BRA, <label>`	*branch to label*
`BRA, <condition code>, <label>`	*if condition code is true, branch to label*
`JMP, <label>`	*jump to label*
`JMP, <condition code>, <label>`	*if condition code is true, jump to label*
`GOTO, <register>`	*jump the address contained in the register*

`BRA` instructions requires one less byte than the equivalent `JMP` instructions, but are limited to branching to a label located at an address -128 bytes or +127 bytes from the next instruction. `JMP` instructions can branch to any address in the function. The `GOTO` instruction jumps to the address specified by the value in a register. If a `BRA`, `JMP`, or `GOTO` instruction specifies an address that is outside the address range of the function, the function will exit. An implicit `RET` instruction is included at the end of all function. `RET` instructions can also be placed within the function.

`RET`	*return from function*
`RET, <condition code>`	*if condition is true, return from function*

Condition Codes

The condition codes used by various instructions are summarized below.

Symbol	Definition	Condition Code	Status Bits
Z, EQ	zero or equal	51	N=0, Z=1
NZ,NE	non-zero or not equal	50	N=0, Z=0
LT	less than	72	N=0, S=1, Z=0
LE	less than or equal	62	(special case)
GT	greater than	70	N=0, S=0, Z=0
GE	greater than or equal	60	(special case)
PZ	plus zero	71	N=0, S=0, Z=1
MZ	minus zero	73	N=0, S=1, Z=1
INF	infinity	C8	I=1, N=0
FIN	finite	C0	I=0, N=0
PINF	plus infinity	E8	I=1, N=0, S=0
MINF	minus infinity	EA	I=1, N=0, S=1
NAN	Not-a-Number	44	N=1
TRUE	always true	00	(special case)
FALSE	always false	FF	(special case)

Labels

Labels must be at the start of a source code line, and must begin with an underscore character, followed by a number or by a sequence of alphanumeric characters, terminated by a colon. Labels are local symbols and are only valid in the function they are defined in. The same label could be used in different functions.

```
_1:
_loop:
_wait:
```

Using Branch Instructions and Labels

The following examples demonstrate the use of branch instructions and labels. Psuedocode and the corresponding FPU assembler code are shown for each example.

If Statement

Psuedocode

```
if tmp < 10
    sum = sum + 1
else
    sum = sum + 10
end if
```

Assembler Code

```
#asm
    SELECTA, tmp                    if tmp < 10
    FCMPI, 10
    BRA, GE, _1

    SELECTA, sum                    sum = sum + 1
    FADDI, 1
    BRA, _2
_1:                                 else
    SELECTA, sum                    sum = sum * 10
    FMULI, 10
_2:                                 endif
```

```
#endasm
```

Repeat Statement

Psuedocode

```
repeat 10 times
    sum = sum + 1
```

Assembler Code

```
#asm
    SELECTA, cnt                    set loop counter to 20
    LSETI, 20

_loop:
    SELECTA, sum                    sum = sum + 1
    FADDI, 1

    LDEC, cnt                       decrement loop counter
    BRA, GT, _loop                  repeat until done
#endasm
```

For Statement

Psuedocode

```
for cnt = startValue to endValue
    sum = sum + 1
next
```

Assembler Code

```
#asm
    SELECTA, cnt                    set loop counter to start value
    LSET, startValue

_loop:
    SELECTA, sum                    sum = sum + 1
    FADDI, 1

    LINC, cnt                       increment loop counter
    LCMP2, cnt, endValue            check for end value
    BRA, LT, _loop                  repeat until done
#endasm
```

String Arguments

Several options are provided for assembler instructions that require a string argument. The simplest form is to use a string constant. The assembler will automatically add a zero terminator as required.

```
STRSET, "test"
```

Special characters can be entered using a backslash followed by two hexadecimal digits.

```
STRSET, "line1\0D\0Aline2"          add carriage return, linefeed between line1 and line2
```

The assembler will also form a string by concatenating multiple string and byte constants.

```
STRSET, "line1", 13, 10, "line2"    results in the same string as above
```

An empty string can be specified in two ways.

```
STRSET, ""
STRSET, 0
```

empty string

Table Instructions

The TABLE, FTABLE, LTABLE, and POLY instructions are only valid inside functions. These instructions specify a count of the number of additional arguments, and the additional arguments are added using the #FLOAT or #LONG directives.

```
TABLE, 4
#FLOAT 10.0
#FLOAT 20.0
#FLOAT 50.0
#FLOAT 100.0
```

load value from table

```
POLY, 3
#float -2.8E-6
#float 0.0405
#float -4.0
```

POLY instruction with coefficients -0.0000028, 0.0405, -4.0

MOP Instruction

The IDE doesn't provide high level support for matrix operations, they must be specified using assembler. There are predefined symbols for the matrix operations that can be used with the **MOP** instruction. For example the following instructions initialize all elements of a 2x2 matrix to 1.0.

```
#asm
   SELECTMA, 10, 2, 2
   LOADBYTE, 1
   MOP, SCALAR_SET
#endasm
```

A list of the predefined symbols for matrix operations are as follows:

```
0     SCALAR_SET
1     SCALAR_ADD
2     SCALAR_SUB
3     SCALAR_SUBR
4     SCALAR_MUL
5     SCALAR_DIV
6     SCALAR_DIVR
7     SCALAR_POW
8     EWISE_SET
9     EWISE_ADD
10    EWISE_SUB
11    EWISE_SUBR
12    EWISE_MUL
13    EWISE_DIV
14    EWISE_DIVR
15    EWISE_POW
16    MULTIPLY
17    IDENTITY
18    DIAGONAL
19    TRANSPOSE
20    COUNT
21    SUM
22    AVE
23    MIN
24    MAX
25    COPY_AB
26    COPY_AC
27    COPY_BA
28    COPY_BC
29    COPY_CA
30    COPY_CB
31    DETERM
32    INVERSE
33    LOAD_RA
34    LOAD_RB
35    LOAD_RC
36    LOAD_BA
37    LOAD_CA
38    SAVE_AR
39    SAVE_AB
40    SAVE_AC
```

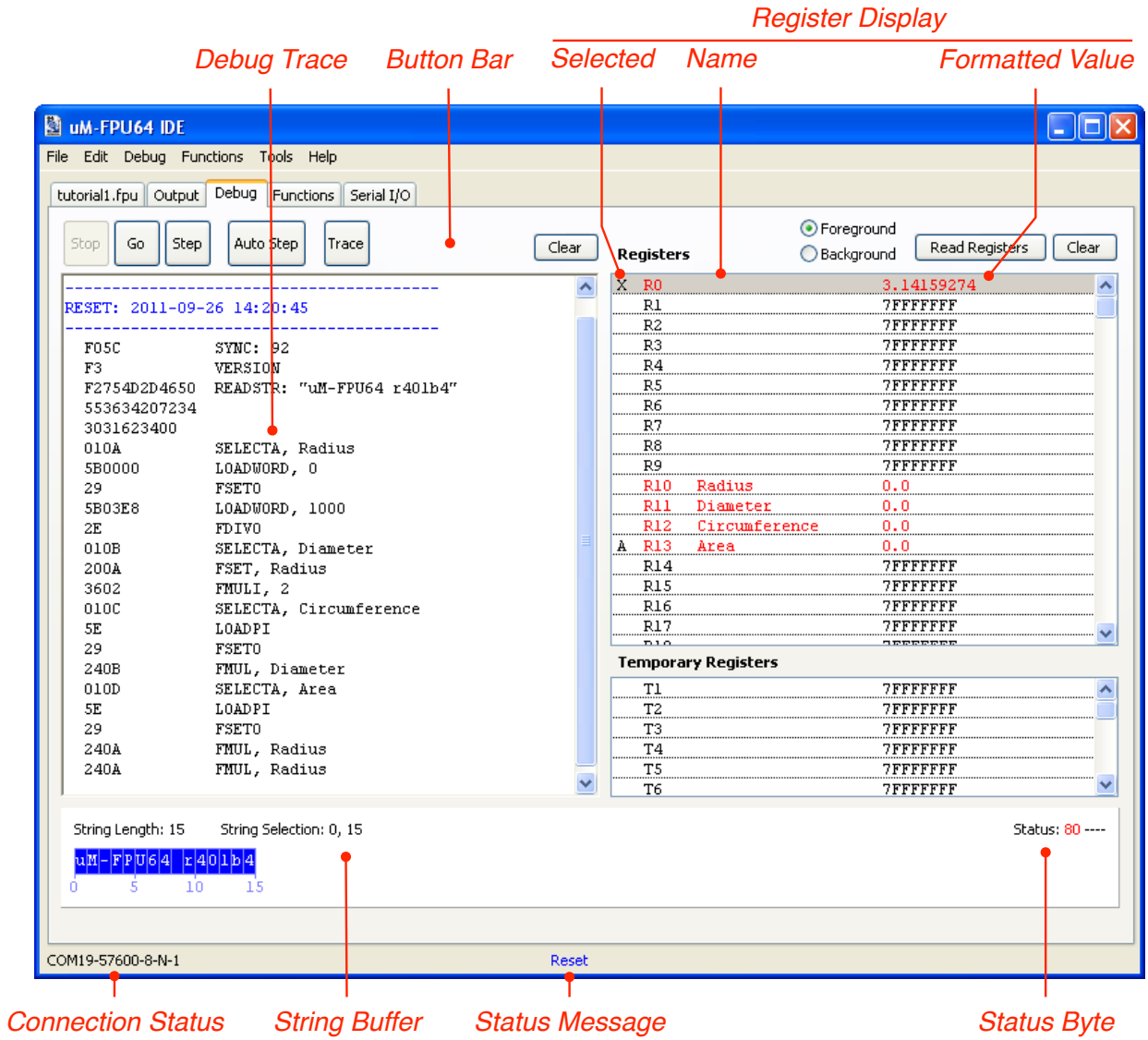
Reference Guide: Debugger

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*.

Debug Window



The ***Debug Trace*** 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 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.

```
   ---------------------------------------
   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
```

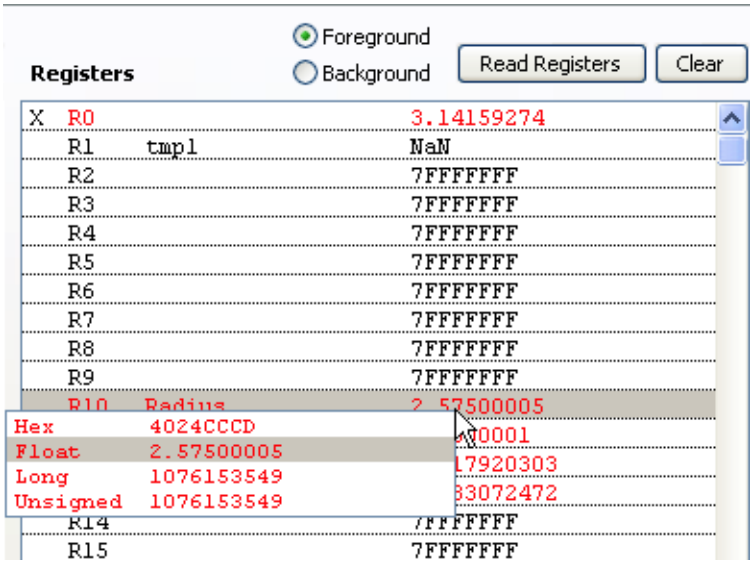
The ***Go***, ***Stop***, and ***Step*** buttons are enabled or disabled depending on the current state of execution. The ***Go*** button is used to continue execution, and is enabled at Reset or after a breakpoint occurs. The ***Stop*** button is used to stop execution after the next FPU instruction 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.

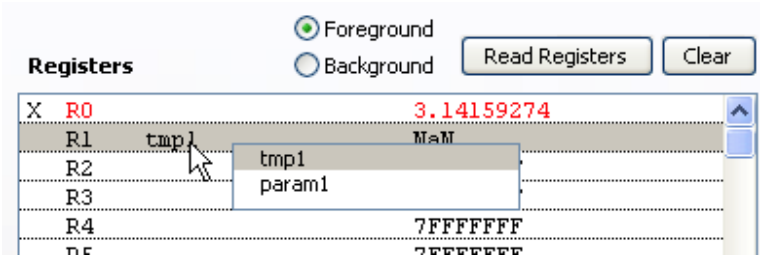
The Register Panel

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.

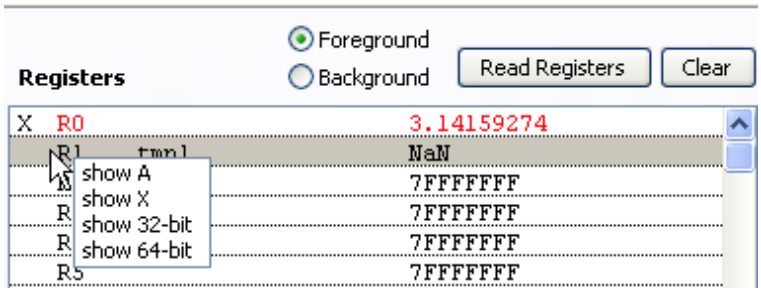
For each register, the register number, optional register name, and formatted value is displayed. If you right-click on the formatted value, a pop-up menu is displayed with the register value displayed in hexadecimal, floating point, long integer, and unsigned long integer format. If you select a different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



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.

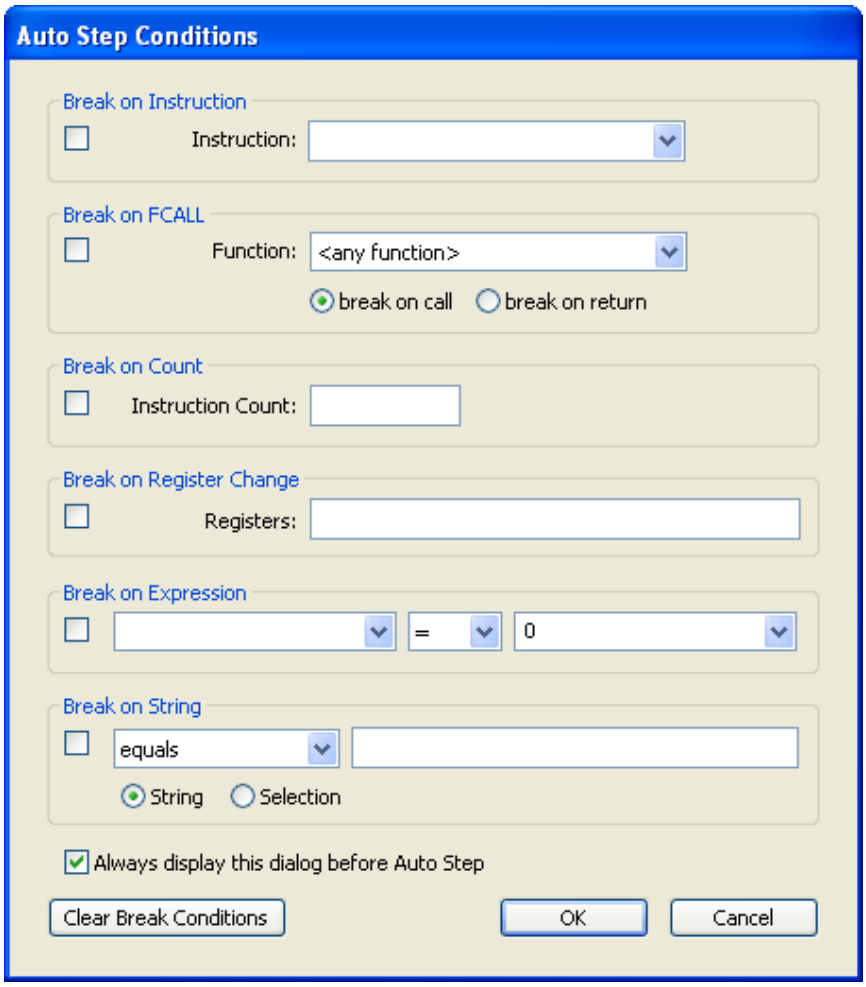
Reference Guide: Auto Step and Conditional Breakpoints

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.

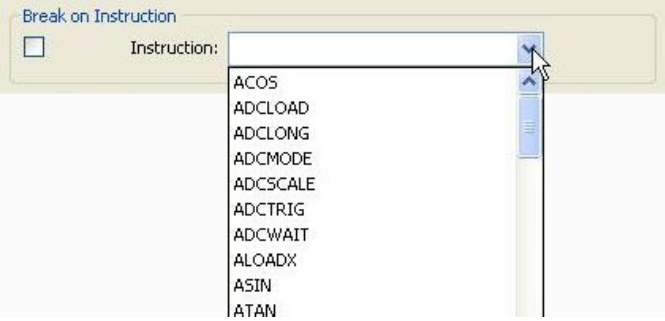
Auto Step Conditions Dialog



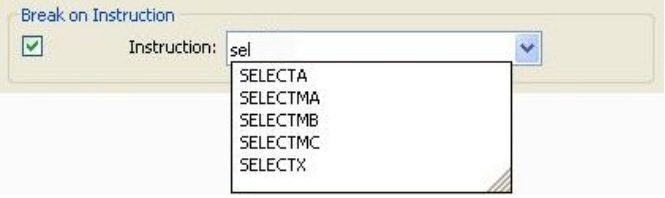
Break on Instruction

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,

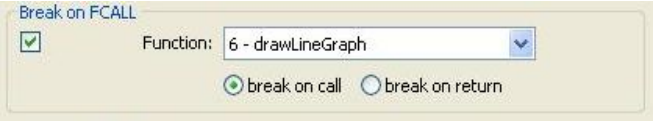


or the opcode can be typed in the field. An auto-complete feature is provided to assist in typing the opcode.

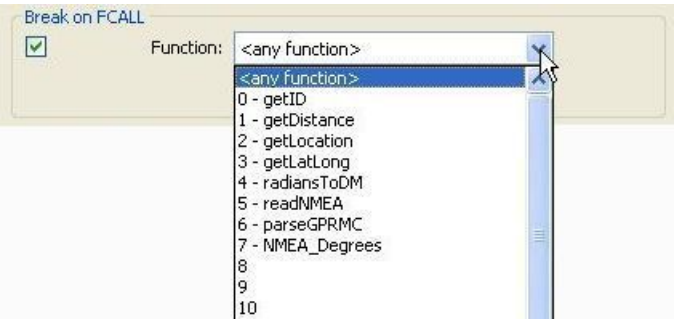


Break on FCALL

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:

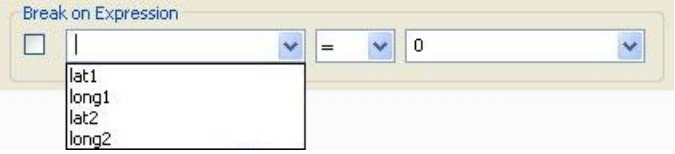
- a single register value (e.g. 1)
- a range of register values (e.g. 3-10 which selects registers 3 through 10)
- an array of register values (e.g. 20:2 which selects two registers starting at registers 20)

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

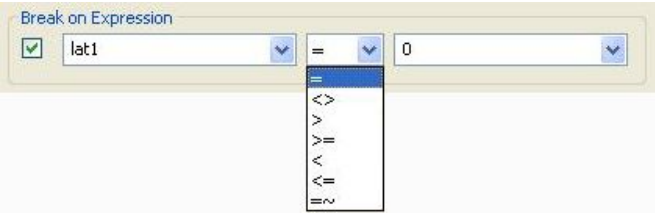
Break on Expression

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.



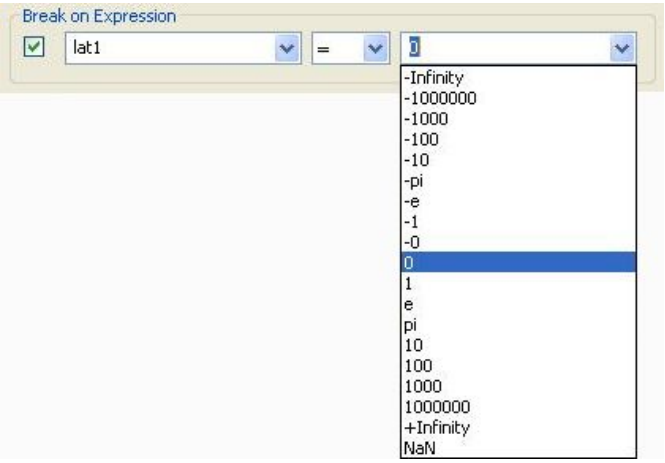
The operator used by the expression is chosen from the middle pop-up menu



The operators are as follows:

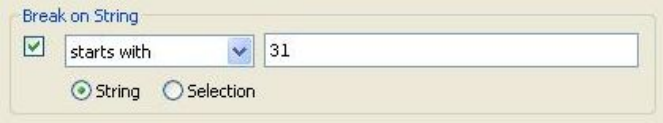
=	equal
<>	not equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
=~	approximately equal

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.

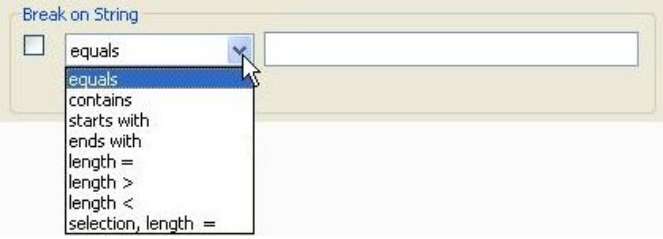


Break on String

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.

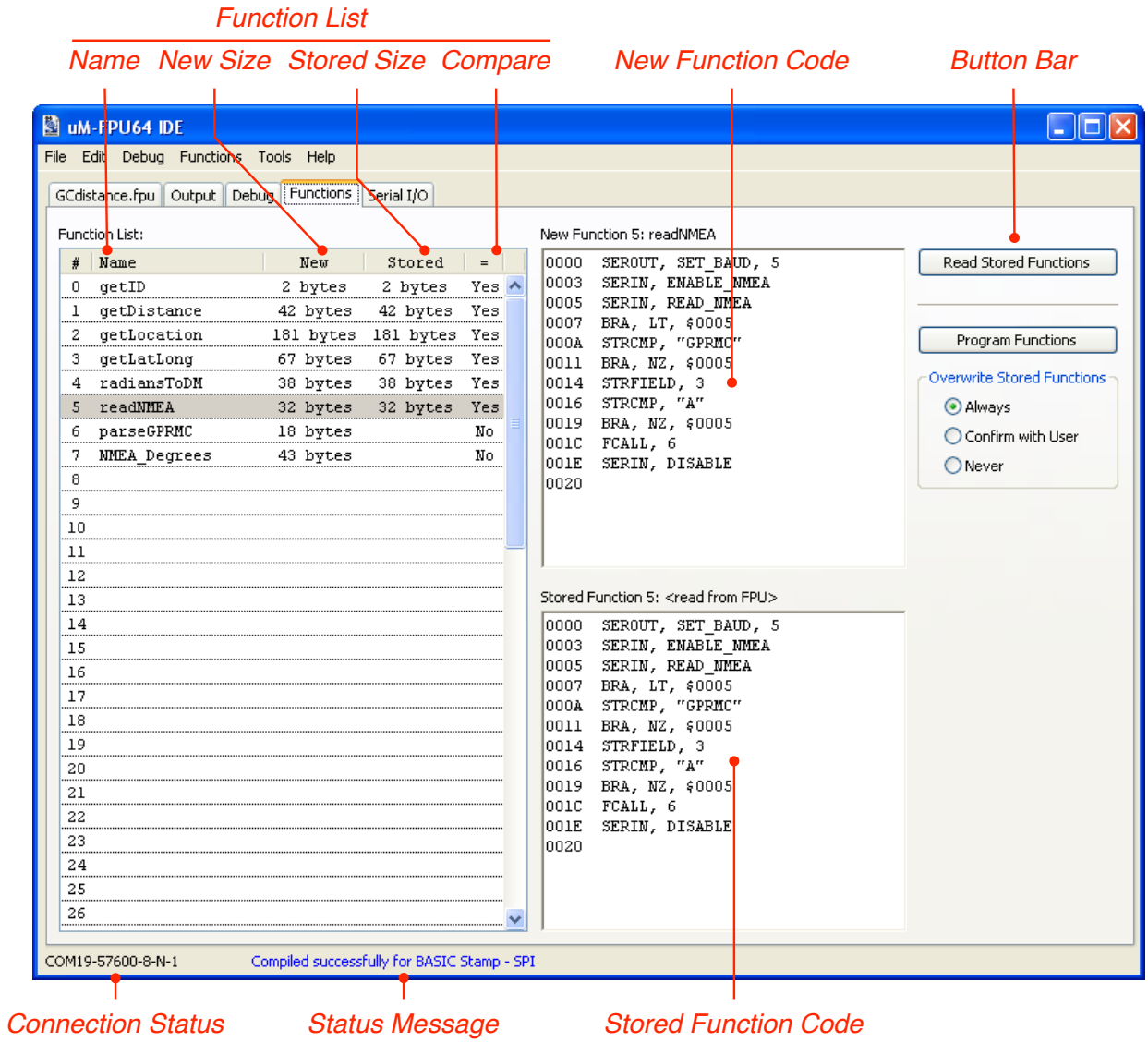


The comparisons for length require a decimal number to be entered in the field on the right. The comparisons for selection, length require two decimal numbers separated by a comma to be entered in the field on the right.

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



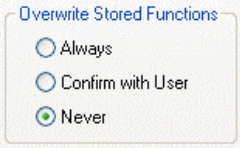
The ***Function List*** provides information about each function defined by the compiler and stored on the FPU. The **Name** column in the ***Function List*** displays the name of all functions defined in the source file. The **New** column shows the size in bytes of the functions defined in the source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

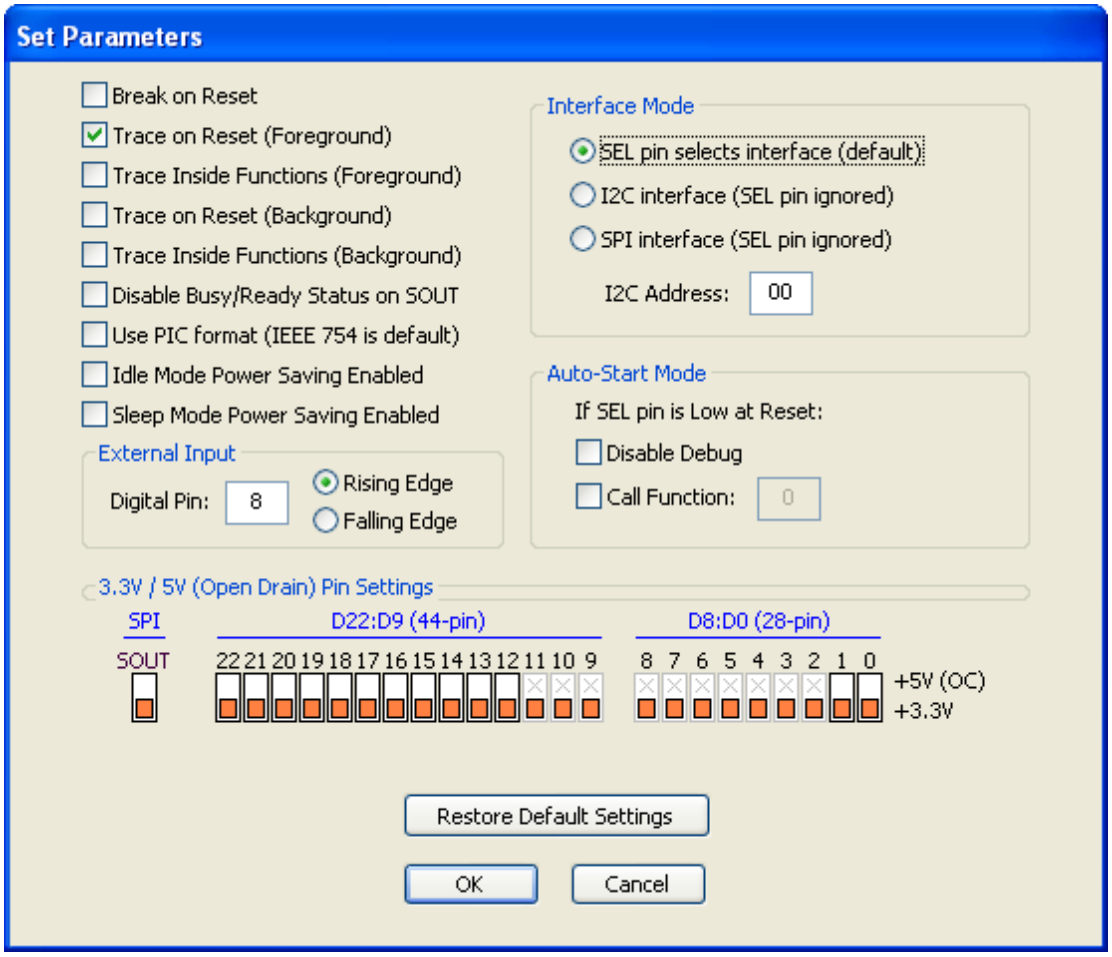
If the **Confirm with User** option is selected, you are asked to confirm whether a new function should replace the previously stored function.

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: Target Description File

Target description files are used to customize the compiler output for a specific microcontroller development language. The IDE supports a wide range of microcontrollers, and a set of predefined target description files are included with the IDE. The system target files are installed and loaded from the following folder:

~\Program Files\Micromega\uM-FPU V3 IDE rxxx\Target Files
(where *rxxx* is the IDE software revision number)

User target files are loaded from the following folder:

\My Documents\Micromega\Target Files

Users can create their own target description files. Target files are text files that can be created and edited with any text editor. The file should then be copied to the user target folder to be loaded when the IDE starts.

The target file contains a series of commands to define how the compiler will generate code for a particular target. To be recognized by the IDE as a target description file, the first line of the file must contain the **TARGET_NAME** command.

A sample target description file is shown below.

```
TARGET_NAME=<Generic C compiler>

; This file defines code generation for a C compiler

MAX_LENGTH=<80>
MAX_WRITE=<6>
TAB_SPACING=<-4>
COMMENT_PREFIX=<//>
SOURCE_PREFIX=<{t}// >
HEX_FORMAT=<0x{byte}>
STRING_HEX_FORMAT=<\x{byte}>

WRITE=<{t}fpu_write{n1}({byte});>
WRITE_BYTE_FORMAT=<{byte}>
WRITE_WORD=<{t}fpu_writeWord({word});>
WRITE_LONG=<{t}fpu_writeLong({long});>
WRITE_FLOAT=<{t}fpu_writeFloat({float});>
WRITE_STRING=<{t}fpu_writeChar("{string}");>
WAIT=<{t}fpu_wait();>

READ_BYTE=<{t}{name} = fpu_read();>
READ_WORD=<{t}{name} = fpu_readWord();>
READ_LONG=<{t}{name} = fpu_readLong();>
READ_FLOAT=<{t}{name} = fpu_readFloat();>

REGISTER_DEFINITION=<#define {name}{t}{register}>
BYTE_DEFINITION=<int {name};>
WORD_DEFINITION=<long {name};>
LONG_DEFINITION=<int32 {name};>
FLOAT_DEFINITION=<float {name};>

PRINT_FLOAT=<{t}print_float({byte});
{t}print_CRLF();>
PRINT_LONG=<{t}print_long({byte});
{t}print_CRLF();>
PRINT_FPUSTRING=<{t}print_fpuString(READSTR);
{t}print_CRLF();>
```

```
PRINT_NEWLINE=<{t}print_CRLF();>
PRINT_STRING=<{t}printf({string});
{t}print_CRLF();>
```

Syntax

The general format of a command is as follows:

```
COMMAND=<ARGUMENT>
```

The name of the command is specified first, followed by an equal sign and the argument surrounded by **< >** characters. The following command defines the target name.

```
TARGET_NAME=<Generic C compiler>
```

Arguments can extend over multiple lines, and have replaceable parameters. Parameters are special keywords surrounded by **{ }** characters. The following command specifies how to write a 16-bit word value to the FPU. The **{byte}** parameter is replaced by the actual value when the code is generated.

```
WRITE_WORD=<  lda    {byte}
    jsr fpu_write
    lda {byte}+1
    jsr fpu_write>
```

Tab Spacing

The <*tab*> character, or `{t}` and `{tn}` parameters, can be used to align the output to particular character positions. They can be inserted into any of the output commands. The <*tab*> character and `{t}` parameter will insert <*space*> characters until the next character position is a multiple of the value specified by the `TAB_SPACING` command. If the value specified by `TAB_SPACING` is positive, only spaces are used to move to the next tab position. If the value is negative, then both <*space*> and <*tab*> used to move to the next tab position. The `{tn}` parameter will insert characters until the character position equals the value specified. If the output is already at a position greater than the character position specified, a single <*space*> or <*tab*> will be output.

Commands

A target description file only needs to contain those commands that are necessary to define the output for a particular target. There are default values for many of the commands. The available commands are as follows:

```
TARGET_NAME
MAX_LENGTH
MAX_WRITE
TAB_SPACING
DECIMAL_FORMAT
HEX_FORMAT
STRING_HEX_FORMAT
OPCODE_PREFIX
COMMENT_PREFIX
SOURCE_PREFIX
SEPARATOR
CONTINUATION

START_WRITE_TRANSFER
START_READ_TRANSFER
STOP_TRANSFER
WAIT
```

```
WRITE
WRITE_BYTE_FORMAT
WRITE_WORD_FORMAT
WRITE_LONG_FORMAT
WRITE_FLOAT_FORMAT
WRITE_STRING_FORMAT

WRITE_BYTE
WRITE_WORD
WRITE_LONG
WRITE_STRING

READ_DELAY
READ_BYTE
READ_WORD
READ_LONG
READ_FLOAT
```

```
REGISTER_DEFINITION                    PRINT_LONG
BYTE_DEFINITION                        PRINT_FPUSTRING
WORD_DEFINITION                        PRINT_NEWLINE
LONG_DEFINITION                        PRINT_STRING
FLOAT_DEFINITION
                                       RESERVED_PREFIX
PRINT_FLOAT                            RESERVED_WORD
```

A detailed description of each command is provided at the end of the section.

Reviewing the Sample File

To better understand target description files, we'll take a closer look at the sample target description file shown at the start of this section.

In order to be recognized as a target description file, the first line of the file must contain the `TARGET_NAME` command. It specifies the name of the target as it will appear in the ***Target Menu*** of the ***Source Window.***

```
TARGET_NAME=<Generic C compiler>
```

The next section defines the maximum output line length, number of bytes per write statement, and prefix characters for comments and hex values.

```
MAX_LENGTH=<80>                        maximum line length of 80 characters
MAX_WRITE=<6>                          maximum of 6 bytes per write statement
TAB_SPACING=<-4>                       use <tab> characters, 4 character per tab
COMMENT_PREFIX=<//>                    comments have // prefix
SOURCE_PREFIX=<{t}// >                 source code has <tab>// prefix
HEX_FORMAT=<0x{byte}>                  hex values have 0x prefix
STRING_HEX_FORMAT=<\x{byte}>           hex string characters have \x prefix
```

The next two commands specify the format for writing out bytes. The `WRITE` command uses three parameters. The `{t}` will be replaced by a *<tab>* character. The `{n1}` is replaced by the number of bytes in the write statement (or the empty string if the write statement has only one byte. The `{byte}` argument is replaced by up to six bytes (set by `MAX_WRITE`). The format for the byte value is determined by the `WRITE_BYTE_FORMAT` command, and is just the value itself with no additional prefix or suffix.

```
WRITE=<{t}fpu_write{n1}({byte});>
WRITE_BYTE_FORMAT=<{byte}>
```

An example of the output generated by these commands is as follows:

```
fpu_write2(SELECTA, temp);
fpu_write(CLRA);
```

Next are the commands for writing out word, long, float and string values. In this example, each of these are defined to use a separate function call. In other cases, the values could be output using the `WRITE` command by defining a a format command instead of a separate function call (i.e. `WRITE_WORD_FORMAT` instead of `WRITE_WORD`).

```
WRITE_WORD=<{t}fpu_writeWord({word});>
WRITE_LONG=<{t}fpu_writeLong({long});>
```

```
WRITE_FLOAT=<{t}fpu_writeFloat({float});>
WRITE_STRING=<{t}fpu_writeChar("{string}");>
```

An example of the output generated by these commands is as follows:

```
fpu_writeWord(1000);
fpu_writeLong(value);
fpu_writeLong(100.25);
fpu_writeString("Result: ");
```

The `WAIT` command specifies the function to call to wait for the FPU ready status.

```
WAIT=<{t}fpu_wait();>
```

The commands for reading data values are shown below.

```
READ_BYTE=<{t}{name} = fpu_read();>
READ_WORD=<{t}{name} = fpu_readWord();>
READ_LONG=<{t}{name} = fpu_readLong();>
READ_FLOAT=<{t}{name} = fpu_readFloat();>
```

An example of the output generated by these commands is as follows:

```
tmp = fpu_read();
cnt = fpu_readWord();
sum = fpu_readLong();
angle = fpu_readFloat();
```

The following command specifies how registers are defined .

```
REGISTER_DEFINITION=<#define {name}{t}{register}>
```

An example of register definitions is as follows:

```
#define angle   10
#define lat1    11
```

Next are the commands to define microcontroller variable.

```
BYTE_DEFINITION=<int {name};>
WORD_DEFINITION=<long {name};>
LONG_DEFINITION=<int32 {name};>
FLOAT_DEFINITION=<float {name};>
```

An example of the output generated by these commands is as follows:

```
int cnt;
long sum;
float angle;
```

Finally, the commands to define print statement.

```
PRINT_FLOAT=<{t}print_float({byte});
{t}print_CRLF();>
```

```
PRINT_LONG=<{t}print_long({byte});
{t}print_CRLF();>
PRINT_FPUSTRING=<{t}print_fpuString(READSTR);
{t}print_CRLF();>
PRINT_NEWLINE=<{t}print_CRLF();>
PRINT_STRING=<{t}printf({string});
{t}print_CRLF();>
```

An example of the output generated by these commands is as follows:

```
print_float(angle);
print_CRLF();
```

Reserved Words

The IDE code generator uses symbolic values for the FPU opcodes. Some microcontroller languages may need a prefix for the opcodes, or some FPU opcodes may conflict with reserved names in the microcontroller language. For example, an object-oriented language like Java requires a module prefix for all constants. The `OPCODE_PREFIX` command can be used to add a prefix to all opcodes.

```
OPCODE_PREFIX=<Fpu.>
```

An example of the opcodes generated is as follows:

```
Fpu.SELECTA
FPU.FADD
```

Other languages may have only a few reserved words that conflict with the FPU opcodes. The RESERVED_WORD command is used to identify these words, and the RESERVED_PREFIX command defines a prefix to make them unique. The following example adds an F_ prefix to three reserved words, the other opcodes would be unaffected.

```
RESERVED_PREFIX=<F_>
RESERVED_WORD=<SIN>
RESERVED_WORD=<COS>
RESERVED_WORD=<TAN>
```

An example of the opcodes generated is as follows:

```
SELECTA
FADD
F_SIN
F_COS
```

Target Description Commands

BYTE_DEFINITION — Define byte variable definition

BYTE_DEFINITION=<*string*>
Default: empty string
Parameters: {byte}
Example:
```
BYTE_DEFINITION=<char {name};>
```

Description: This command defines the instruction sequence used to define an 8-bit integer variable. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

COMMENT_PREFIX — Set the prefix for comments

COMMENT_PREFIX=<*string*>
Default: ; (semi-colon)
Parameters: none
Example:
```
COMMENT_PREFIX=<//>
```

Description: This command defines the prefix characters used before a comment.

CONTINUATION — Define line continuation for WRITE command

CONTINUATION=<*string*>
Default: empty string
Parameters: none
Example:
```
CONTINUATION=< _
       >
```

Description: This command sets the continuation sequence used for continuing the WRITE command instructions on multiple lines. If the CONTINUATION command is set to an empty string, no line continuation is allowed.

DECIMAL_FORMAT — Set the prefix for decimal numbers

DECIMAL_FORMAT=<*string*>
Default: empty string
Parameters: {byte}
Example:
```
DECIMAL_FORMAT=<.{byte}>
```

Description: This command sets the prefix character for decimal numbers.

FLOAT_DEFINITION — Define float variable definition

FLOAT_DEFINITION=<*string*>
Default: empty string
Parameters: {name}
Example: `FLOAT_DEFINITION=<float {name};>`

Description: This command defines the instruction sequence used to define a 32-bit floating point variable. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

HEX_FORMAT — Set the prefix for hexadecimal numbers

HEX_FORMAT=<*string*>
Default: $ (dollar sign)
Parameters: {byte}
Example: `HEX_FORMAT=<0x{byte}>`

Description: This command sets the prefix character for hexadecimal numbers.

LONG_DEFINITION — Define long variable definition

LONG_DEFINITION=<*string*>
Default: empty string
Parameters: none
Example: `LONG_DEFINITION=<long {name};>`

Description: This command defines the instruction sequence used to define a 32-bit integer variable. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

MAX_LENGTH — Set maximum length of write instruction

MAX_LENGTH=<*length*>
Default: 80
Parameters: none
Example: `MAX_LENGTH=<90>`

Description: This command defines the maximum length of a source line.

MAX_WRITE — Set maximum number of bytes in write instruction

MAX_WRITE=<*n*>
Default: 1
Parameters: none
Example: `MAX_WRITE=<8>`

Description: This command defines the maximum number of bytes in a write command.

OPCODE_PREFIX — Set the prefix for opcodes in WRITE command

OPCODE_PREFIX=<*string*>
Default: empty string
Parameters: none
Example:
```
OPCODE_PREFIX=<FPU_>
```

Description: This command sets the prefix for opcodes used in write_command. It can be used in conjunction with a symbol definition file to ensure unique names for the opcode constants.

PRINT_FLOAT — Define instructions to print float value

PRINT_FLOAT=<*string*>
Default: empty string
Parameters: {byte}
Example:
```
PRINT_FLOAT=<format = {byte}
GOSUB PRINT_FLOAT>
```

Description: This command defines the instruction sequence to print a 32-bit floating point value. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

PRINT_FPUSTRING — Define instructions to print FPU string

PRINT_FPUSTRING=<*string*>
Default: empty string
Parameters: none
Example:
```
PRINT_FPUSTRING=<GOSUB PRINT_FPUSTRING>
```

Description: This command defines the instruction sequence to print FPU string. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

PRINT_LONG — Define instructions to print long value

PRINT_LONG=<*string*>
Default: empty string
Parameters: {byte}
Example:
```
PRINT_FLOAT=<format = {byte}
GOSUB PRINT_LONG>
```

Description: This command defines the instruction sequence to print a 32-bit integer value. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

PRINT_NEWLINE **Define instructions to print new line**

PRINT_NEWLINE=<*string*>
Default: empty string
Parameters: none
Example: `PRINT_NEWLINE=<DEBUG CR>`

Description: This command defines the instruction sequence to print a new line. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

PRINT_STRING **Define instructions to print text string**

PRINT_STRING=<*string*>
Default: empty string
Parameters: {string}
Example: `PRINT_STRING=<DEBUG "{string}">`

Description: This command defines the instruction sequence to print text string. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

READ_BYTE **Define instructions to read 8-bit value**

READ_BYTE=<*string*>
Default: empty string
Parameters: none
Example: `READ_BYTE=<{name} = fpu_readByte();>`

Description: This command defines the instruction sequence to use to read an 8-bit value. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

READ_DELAY **Define instructions for read delay**

READ_DELAY=<*string*>
Default: empty string
Parameters: none
Example: `READ_DELAY=<call fpu_readDelay();>`

Description: This command defines the instruction sequence to be used to wait for the read delay. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

READ_LONG **Defines command to read 32-bit value**

READ_LONG=<*string*>
Default: empty string
Parameters: none
Example: `READ_LONG=<{name} = fpu_readLong();>`

Description: This command defines the instruction sequence to use to read a 32-bit value. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

READ_WORD **Defines instructions to read 16-bit value**

READ_WORD=<*string*>
Default: empty string
Parameters: none
Example: `READ_WORD=<{name} = fpu_readWord();>`

Description: This command defines the instruction sequence to use to read a 16-bit value. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

REGISTER_DEFINITION **Define register definition**

REGISTER_DEFINITION=<*string*>
Default: empty string
Parameters: {name}, {register}
Example: `REGISTER_DEFINITION=<#define {name}  {register}>`

Description: This command defines the instruction sequence used to define a register constant. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

RESERVED_PREFIX **Define prefix for reserved words**

RESERVED_PREFIX=<*string*>
Default: F_ (F and underscore)
Parameters: none
Example: `RESERVED_PREFIX=<FPU_>`

Description: This command defines the prefix to add to reserved words in order to make them unique.

RESERVED_WORD **Define reserved word**

RESERVED_WORD=<*string*>
Default: empty string
Parameters: none
Example: `RESERVED_WORD=<SIN>`

Description: This command defines a reserved word. Multiple RESERVED_WORD commands can be used, with each command specifying one reserved word.

SEPARATOR — Define separator character for WRITE command

SEPARATOR=<*string*>
Default: , (comma and space)
Parameters: none
Example: `SEPARATOR=<, >`

Description: This command sets the separator character used between items in write_command.

SOURCE_PREFIX — Set indent for the start of a comment line

SOURCE_PREFIX=<*string*>
Default: ; (semi-colon)
Parameters: none
Example: `SOURCE_PREFIX=<    ;-->`

Description: This command sets the prefix that's added to source code lines that are copied as comments included with the generated code. The correct string must be specified for a valid comment.

START_READ_TRANSFER — Define instructions for start of a read transfer

START_READ=<*string*>
Default: empty string
Parameters: none
Example: `START_READ=<CALL START_READ();>`

Description: This command defines the instruction sequence used to start a read transfer. Some implementations will not require this command. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

START_WRITE_TRANSFER — Define instructions for start of a write transfer

START_WRITE=<*string*>
Default: empty string
Parameters: none
Example: `START_WRITE=<CALL START_WRITE();>`

Description: This command defines the instruction sequence used to start a write transfer. Some implementations will not require this command. A <*carriage return*> and <*linefeed*> character is appended to the end of the output.

STOP_TRANSFER — Define instructions for end of read or write transfer

STOP=<*string*>
Default: empty string
Parameters: none

Example: `STOP=<CALL STOP();>`

Description: This command defines the instruction sequence used to end a read or write transfer. Some implementations will not require this command. A <*carriage return*> and <*linefeed*> character is appended to the end of the output.

STRING_HEX_FORMAT — Define format for non-printable string characters

STRING_HEX_FORMAT=<*string*>
Default: empty string
Parameters: none
Example: `STRING_HEX_FORMAT=<\{byte}>`

Description: This command defines the syntax for writing a non-printable character using write_command.

TAB_SPACING — Set number of characters per tab

TAB_SPACING=<*n*>
Default: 4
Parameters: none
Example: `TAB_SPACING=<8>`

Description: This command sets the number of characters in a tab. The absolute value of *n* specifies the number of characters. If *n* is positive, only spaces are used to move to the next tab position. If *n* is negative, then horizontal tabs (0x09) and spaces are used to move to the next tab position.

TARGET_NAME — Define the target name

TARGET_NAME=<*target name*>
Default: none
Parameters: none
Example: `TARGET_NAME=<C compiler>`

Description: This command must be on the first line of the file in order for the file to be recognized as a target description file. It defines the name that will appear in the target menu.

WAIT — Define instructions to wait for ready status

WAIT=<*string*>
Default: empty string
Parameters: none
Example: `WAIT=<call fpu_wait();>`

Description: This command defines the instruction sequence used to wait for the FPU ready status. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

WORD_DEFINITION — Define word variable definition

WORD_DEFINITION=<*string*>
Default: empty string
Parameters: {name}
Example: `WORD_DEFINITION=<int {name};>`

Description: This command defines the instruction sequence used to define a 16-bit integer variable. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

WRITE — Define instructions to write bytes

WRITE=<*string*>
Default: empty string
Parameters: {byte}
Example: `WRITE=<call fpu_write({byte});>`

Description: This command defines the instruction sequence used to write bytes to the FPU, and is required for all implementations. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

WRITE_BYTE — Define instructions to write 8-bit value

WRITE_BYTE=<*string*>
Default: empty string
Parameters: none
Example: `WRITE_BYTE=<call fpu_write({byte});>`

Description: This command defines the instruction sequence used to output an 8-bit value. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

WRITE_BYTE_FORMAT — Define 8-bit value format for WRITE command

WRITE_BYTE_FORMAT=<*string*>
Default: empty string
Parameters: {byte}
Example: `WRITE_BYTE_FORMAT=<{byte}>`

Description: This command defines the syntax for writing an 8-bit value using the WRITE command.

WRITE_LONG — Define instructions to write 32-bit value

WRITE_LONG=<*string*>
Default: empty string
Parameters: none
Example: `WRITE_LONG=<call fpu_writelong({long});>`

Description: This command defines the instruction sequence used to output a 32-bit value. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

WRITE_LONG_FORMAT **Define 32-bit value format for WRITE command**

WRITE_LONG=<*string*>
Default: empty string
Parameters: none
Examples:
```
WRITE_LONG=<{byte}<<24, {byte}<<16, {byte}<<8, {byte}>
WRITE_LONG=<{word}(1), {word}(2)>
WRITE_LONG=<{long}>
```

Description: This command defines the syntax for writing a 32-bit value using the WRITE command.

WRITE_WORD **Define instructions to write 16-bit value**

WRITE_WORD=<*string*>
Default: empty string
Parameters: none
Example:
```
WRITE_WORD=<call fpu_writeWord{word});>
```

Description: This command defines the instruction sequence used to output a 16-bit value. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

WRITE_WORD_FORMAT **Define 16-bit value format for WRITE command**

WRITE_WORD=<*string*>
Default: empty string
Parameters: {byte}, {word}
Examples:
```
WRITE_WORD=<{word}\16>
WRITE_WORD=<{byte}<<8, {byte}>
```

Description: This command defines the syntax for writing a 16-bit value using the WRITE command.

WRITE_STRING **Define instructions to write string value**

WRITE_STRING=<*string*>
Default: empty string
Parameters: none
Example:
```
WRITE_STRING=<call fpu_writeString("{string}");>
```

Description: This command defines the instruction sequence used to output a zero-terminated string value. A <*carriage return*> and <*linefeed*> is appended to the end of the output.

WRITE_STRING_FORMAT **Define write string format for WRITE command**

WRITE_STRING=<*string*>
Default: empty string
Parameters: none
Example: `WRITE_STRING=<"{string}">`

Description: This command defines the syntax for writing a a zero-terminated string using the WRITE command.