



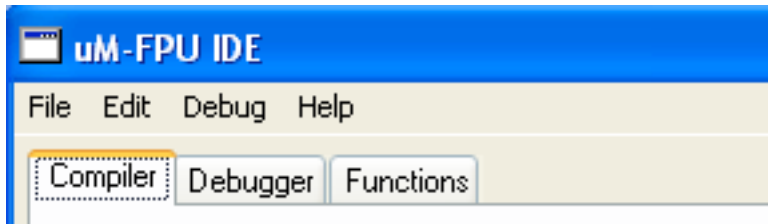
Using the uM-FPU Integrated Development Environment

Micromega Corporation

Introduction

The integrated development environment for the uM-FPU provides an easy-to-use tool for developing applications using the uM-FPU floating point coprocessor and also facilitates the use of the advanced features of the chip. It provides an expression compiler that generates code for various target platforms, a debugger to support development and testing of uM-FPU code, and a function programmer that allows the user to store functions on the uM-FPU, which can greatly reduce memory usage on the microcontroller and significantly increase speed of operation.

The three main modes of operation are each represented by a separate window in the application. There are three tabs located at the top left of the window. Clicking on one of these tabs will display the Compiler, Debugger, or Functions window (see figure below).

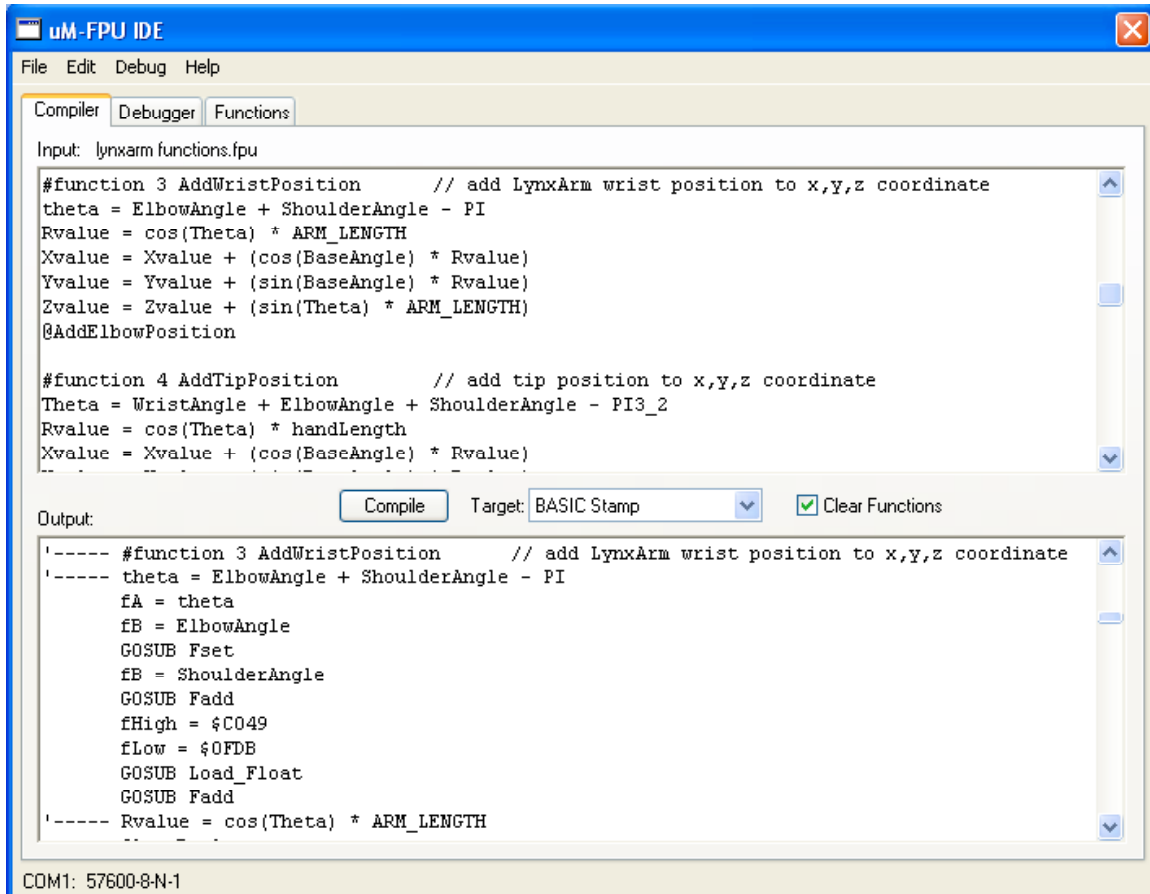


Each of these main windows is described in detail below.

Compiler

The expression compiler allows the user to specify expressions in common math notation (e.g. $x = \cos(\theta) * \text{length}$). The compiler will automatically generate the necessary uM-FPU code for the selected target platform. Support is currently provided for uM-FPU Opcodes, Basic Stamp, Javelin Stamp, and PICAXE. More targets will be added as other microcontrollers are supported.

The figure below shows an example of the Compile window.



To use the compiler, the user enters expressions in the Input field, selects the desired target from the Target: uM-FPU Opcodes pop-up menu and presses the Compile button. The compiled code is generated and displayed in the Output field. The compiler output can be easily copy-and-pasted to the user's main program in the microcontroller development environment. The Open... menu item in the File menu can be used to open files in the Input field and the Save and Save As... menu items can be used to save files. If an error occurs during compile, an error message is displayed and the location of the error is highlighted.

Expressions

Expression can contain uM-FPU registers, microcontroller variables, constants, math operators, math functions and parentheses. Expressions evaluate left to right with no operator precedence. Type conversion between the left and right sides is handled automatically.

The following register definitions are built-in:

- F0 ... F15 specifies that register 0 to 15 contains a floating point value
- L0 ... L15 specifies that register 0 to 15 contains a long integer value

U0 ... U15 specifies that register 0 to 15 contains an unsigned long integer value

The following constants are built-in:

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

The following math functions are built-in:

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

For example, the expression

F1 = F2 + F3 * F4

would generate the following code depending on the selected target:

uM-FPU	BASIC Stamp	Javelin Stamp	PICAXE
Opcodes	fA = 1	f1.set(f2)	reg = 1
SELECTA R1	FB = 2	f1.add(f3)	opcode = SELECTA
SET R2	GOSUB Fset	f1.multiply(f4)	gosub fpu_command
FADD R3	FB = 3		reg = 2
FMUL R4	GOSUB Fadd		opcode = SET
	FB = 4		gosub fpu_command
	GOSUB Fmultiply		reg = 3
			opcode = FADD
			gosub fpu_command
			reg = 4
			opcode = FMUL
			gosub fpu_command

The expression

F1 = 2 * F2 + 5

will generate the following BASIC Stamp code:

```
fA = 1
fHigh = $4000          (Note: 32-bit floating point value for 2)
fLow = $0000
GOSUB Load_Float
GOSUB Fset
fB = 2
GOSUB Fmultiply
fHigh = $40A0         (Note: 32-bit floating point value for 5)
fLow = $0000
GOSUB Load_Float
GOSUB Fadd
```

Symbolic Names

Using symbolic names can make expressions easier to read and understand. Symbolic names can be defined for registers, microcontroller variables, or constants using the EQU operator. Registers are defined using one of the built-in definitions or using a previously defined definition.

e.g.

```
Y EQU F1
X EQU F2
Y = 2 * X + 5
```

Constants

Constants can be defined for use in expressions.

e.g.

```
LENGTH EQU 4.75
```

Optionally, the CON operator can be used to define constants. The CON operator is equivalent to the EQU operator but is restricted to constants.

e.g.

```
LENGTH CON 4.75
```

The compiler simplifies constant expressions to generate a single constant value. For example, the expression

```
Phase2 = Angle * PI / 2
```

will generate the following BASIC Stamp code:

```
fA = Phase2
fB = Angle
GOSUB Fset
fHigh = $3FC9           (Note: 32-bit floating point value for PI / 2)
fLow = $0FDA
GOSUB Load_Float
GOSUB Fmultiply
```

Microcontroller Variables

Microcontroller variables can be defined using the EQU operator and one of the following keywords:

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

e.g.

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

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

e.g.

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

fA = degreesF           (degrees Fahrenheit will be stored in register 1)
GOSUB Left
fLow.LOWBYTE = degreesC (degrees centigrade loaded from microcontroller)
GOSUB Load_FloatByte
GOSUB Fset
fHigh = $3FE6           (multiply by 9 / 5)
fLow = $6666
GOSUB Load_Float
GOSUB Fmultiply
GOSUB Right
GOSUB Fset
fHigh = $4200           (add 32)
fLow = $0000
GOSUB Load_Float
GOSUB Fadd
```

Optionally, the VAR operator can be used to define microcontroller variables. The VAR operator is equivalent to the EQU operator but is restricted to variables.

e.g.

```
count VAR BYTE
```

Comments

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

Order of Evaluation

Expressions are evaluated from left to right with no operator precedence, but constant expressions are first reduced to single constant values. In some cases it may be necessary to use parenthesis to get the desired result. For example, $F1 = F2 * 2 + 5$ is evaluated as $F1 = F2 * 7$ because the constant expression $2 + 5$ is evaluated first. If the desired result is to first multiply $F2$ by 2 then add 5, you would need to break up the constant expression by writing $F1 = 2 * F2 + 5$ or by using parentheses.

e.g.

```
F1 = (F2 * 2) + 5.
```

Parenthesis can also be used in other situations where the order of evaluation needs to be changed. There can be up to five levels of parenthesis used. One level is automatically used if the value on the left side of the equation is also used on the right side of the equation other than as the first operand. For example, the expression

```
X = Y - X
```

requires a temporary value and will generate the following BASIC Stamp:

```
fA = X
GOSUB Left
fB = Y
GOSUB Fset
fB = X
GOSUB Fsubtract
GOSUB Right
GOSUB Fset
```

Whereas the expression

```
X = X - Y
```

which doesn't need a temporary value will generate the following BASIC Stamp code:

```
fA = X
fB = Y
GOSUB Fsubtract
```

Functions

Stored functions are specified using the #FUNCTION directive. After a #FUNCTION directive is encountered, all compiled code is targeted for the uM-FPU and is directed to the appropriate function defined in the Functions window. This continues until another #FUNCTION directive is encountered, an #END directive is encountered, or the end of input is reached. The #FUNCTION directive can optionally include a function name that can be used in the remainder of the file.

e.g.

```
#FUNCTION functionName [functionName]
```

where:

functionNumber	is a value between 0 and 63
functionName	is a symbol name associated with this function

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

e.g.

```
@functionNumber
```

where:

```
functionNumber      is a value between 0 and 63
```

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

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

#FUNCTION 0 Hypotenuse
Z = SQRT(X*X + Y*Y)
#END

X = Value1
Y = Value2
@Hypotenuse
```

If a function is called from inside another function, execution will not return to the original function (i.e. it is a GOTO not a GOSUB). This can still be useful to chain together functions. For example, if you were updating the position of a robotic arm, you could chain through relative offsets of each joint to get the cumulative offset.

e.g.

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

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

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

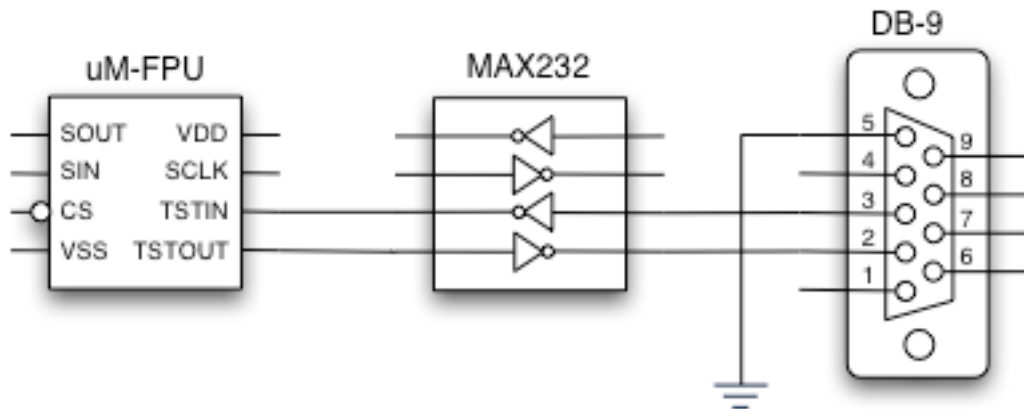
The Clear Functions checkbox will clear the function list before compiling. If the box is not checked and a #FUNCTION directive is encountered that defines a function that is already defined in the function list an error message will be displayed.

Debugger

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

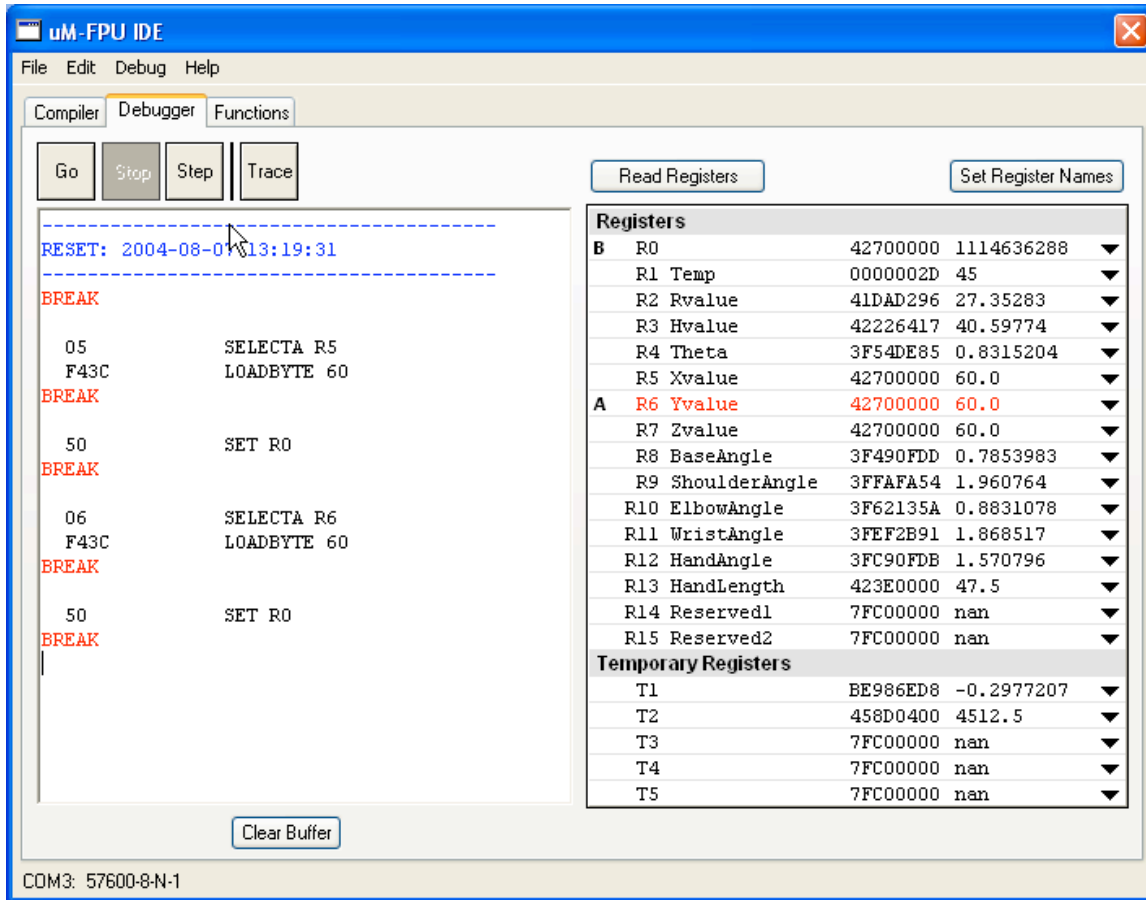
Connecting the Debugger

The built-in debugger operates through a serial connection using the TSTIN and TSTOUT pins of the uM-FPU. The connection to a standard PC serial port is shown in the figure below. The serial connection must be configured as 57,600 baud, 8 bits, no parity, one stop bit.



Debug Window

The figure below shows an example of the Debug window.



The Go/Stop/Step/Trace buttons at the top left of the debugger window control the break and trace features. The scrolling window on the left of the window displays trace messages. The panel on the right displays the contents of the uM-FPU registers. Connection status is displayed at the lower left of the IDE window. If the port needs to be changed use the Select Port... menu item in the Debug menu.

Trace Messages

Trace messages are displayed in a scrolling window on the left of the debug window. When a reset occurs a message is displayed showing the date and time of the reset.

e.g.

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

If a breakpoint occurs, the last uM-FPU instruction executed before the breakpoint is displayed, followed by the BREAK message.

e.g.

```
05          SELECTA R5
F43C        LOADBYTE 60
BREAK
```

If tracing is enabled, all uM-FPU instructions are displayed as they are executed.

e.g.

```
TRACE: ON
```



```

0A          SELECTA R10
F55A       LOADUBYTE 90
50         SET R0
EF         TORADIANS
303FC90FDB WRITEB R0: 0x3FC90FDB
0C        SELECTA R12
50        SET R0
...       ...

```

See Appendix A for a uM-FPU opcode summary.

Breakpoint and Trace Buttons

Breakpoints and tracing are controlled with the following buttons:



The Go/Stop/Step buttons are enabled or disabled depending on the current state of execution. The Go button is enabled after a breakpoint and is used to continue execution. The Stop button is used to cause a breakpoint after the next uM-FPU instruction is executed. If the uM-FPU is idle when the Stop button is pressed the breakpoint will not occur until the next uM-FPU instruction is executed. If the uM-FPU is already at a breakpoint, then the Stop button will be disabled. The Step button is used to single step through the instructions, a new breakpoint occurs after each instruction. (Note: The SELECTA and SELECTB opcodes will not cause a breakpoint because these opcodes do not require a busy/wait check before issuing the next opcode.)

Pressing the button will clear the contents of the trace window.

Register Panel

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

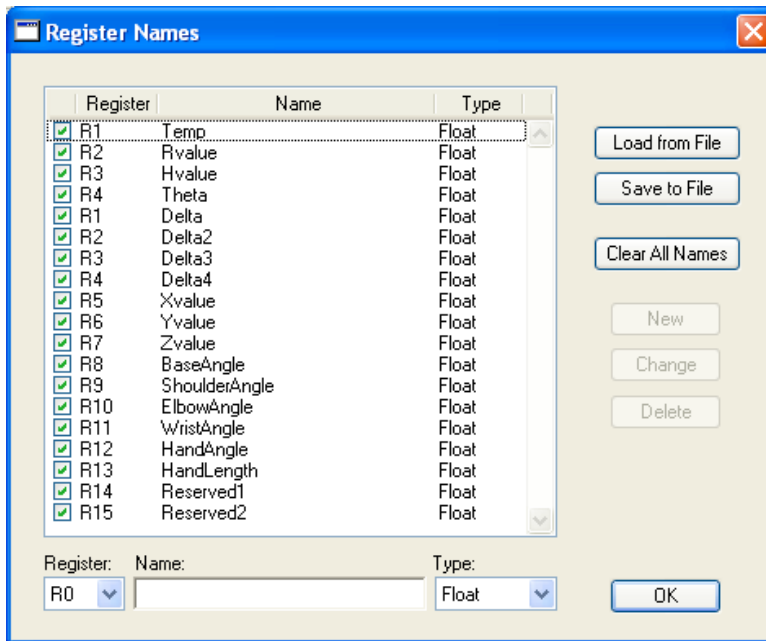
e.g.



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

Register Names

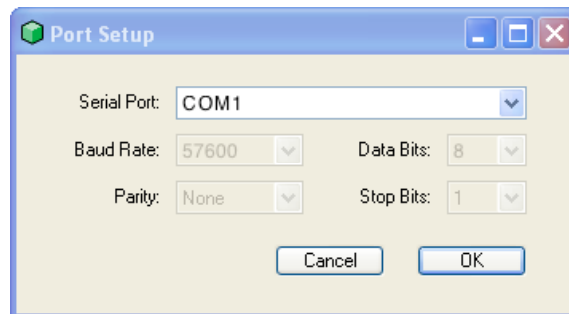
Register names are automatically set by any register definitions that are compiled, but they can also be specified manually by pressing the button. The following dialog is displayed:



New register names are defined by selecting the register from the Register pop-up menu at the bottom of the dialog, entering the name in the Name field, selecting the display type from the Type pop-up menu and pressing the **New** button. A definition can be changed by selecting a definition in the scrolling list, making the changes using the Register pop-up, Name field and Type pop-up, and pressing the **Change** button. A definition can be deleted by selecting the definition in the scrolling list and pressing the **Delete** button. All names can be cleared by pressing the **Clear All Names** button. Register names can be loaded from a file by pressing the **Load from File** button and saved to a file by pressing the **Save to File** button. The checkbox at the left of the list can be used to enable or disable register definitions. This can be useful when debugging programs that have multiple definitions for the same register.

Debug Menu

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



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

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

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

The Break on Reset menu item will automatically cause a breakpoint after reset.

Note: The Trace on Reset and Break on Reset features work by watching for a reset and sending commands sent to the built-in debugger. In some cases, after a reset, the program may have time to execute some uM-FPU instructions before the debug command is received. If it is necessary to see all of the instructions after a reset then a TRACEBRK or TRACEON instruction should be inserted into the code.

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

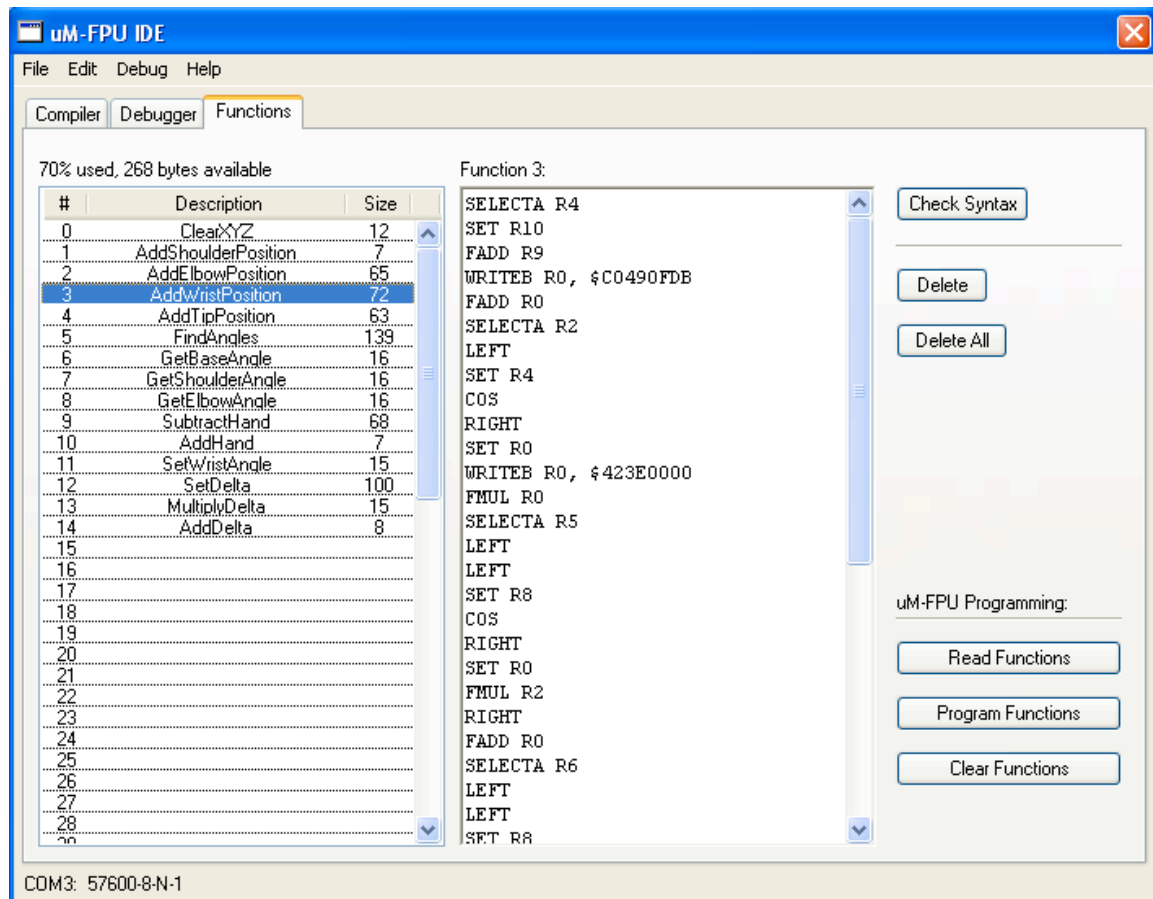
The Read Version menu item will display the version of the uM-FPU in the trace window.

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

Functions

The Functions window provides support for storing functions on the uM-FPU. This can greatly reduce memory usage on the microcontroller and significantly increase speed of operation. The uM-FPU reserves 1024 bytes of flash memory for storing up to 64 functions. Functions are stored as a string of uM-FPU instructions. Functions can be entered or modified directly in the function window if desired, but the most effective way to define functions is to use the #FUNCTION directive in the Compiler.


The figure below shows an example of the Function window.





The status message at the top left of the window shows the percentage of function memory currently used on the uM-FPU, and the number of bytes available. The scrolling list on the left shows all of the currently defined functions. For each function, the name of the function and its size in bytes is displayed. Functions are selected by clicking on an item in the list. The center field shows the instructions for the currently selected function.

Pressing the button checks the syntax of the currently selected function. If an error is detected, a message will be displayed and the location of the error will be highlighted. Syntax is automatically checked before selected a new function or programming the function.

To delete a function from the function list, select the function in the list and press the button. Pressing the button deletes all functions from the function list.

Pressing the  button reads the currently stored function from the uM-FPU and stores them in the function list.

Pressing the  button programs the uM-FPU with all of the functions in the function list. If a function is currently stored on the uM-FPU and no new definition is specified in the function list, the currently stored function will be retained.

Pressing the  button will clear all of the stored functions on the uM-FPU.

Contact Information

See the Micromega Corporation website at: <http://www.micromegacorp.com/>

Appendix A

uM-FPU Opcode Summary

Opcode Name	Data Type	Opcode	Arguments	Returns	B Reg	Description
SELECTA		0x				Select A register
SELECTB		1x			x	Select B register
WRITEA	Either	2x	yyyy zzzz			Write register and select A
WRITEB	Either	3x	yyyy zzzz		x	Write register and select B
READ	Either	4x		yyyy zzzz		Read register
SET	Either	5x				$A = B$
FADD	Float	6x			x	$A = A + B$
FSUB	Float	7x			x	$A = A - B$
FMUL	Float	8x			x	$A = A * B$
FDIV	Float	9x			x	$A = A / B$
LADD	Long	Ax			x	$A = A + B$
LSUB	Long	Bx			x	$A = A - B$
LMUL	Long	Cx			x	$A = A * B$
LDIV	Long	Dx			x	$A = A / B$
SQRT	Float	E0				$A = \text{sqrt}(A)$
LOG	Float	E1				$A = \ln(A)$
LOG10	Float	E2				$A = \log(A)$
EXP	Float	E3				$A = e ** A$
EXP10	Float	E4				$A = 10 ** A$
SIN	Float	E5				$A = \sin(A)$ radians
COS	Float	E6				$A = \cos(A)$ radians
TAN	Float	E7				$A = \tan(A)$ radians
FLOOR	Float	E8				$A = \text{nearest integer } \leq A$
CEIL	Float	E9				$A = \text{nearest integer } \geq A$
ROUND	Float	EA				$A = \text{nearest integer to } A$
NEGATE	Float	EB				$A = -A$
ABS	Float	EC				$A = A $
INVERSE	Float	ED				$A = 1 / A$
DEGREES	Float	EE				Convert radians to degrees $A = A / (\text{PI} / 180)$
RADIANS	Float	EF				Convert degrees to radians $A = A * (\text{PI} / 180)$
SYNC		F0		5C		Synchronization
FLOAT	Long	F1			0	Copy A to Register 0 Convert long to float
FIX	Float	F2			0	Copy A to Register 0 Convert float to long
FCOMPARE	Float	F3		ss		Compare A and B (floating point)
LOADBYTE	Float	F4	bb		0	Write signed byte to Register 0

Opcode Name	Data Type	Opcode	Arguments	Returns	B Reg	Description
						Convert to float
LOADUBYTE	Float	F5	bb		0	Write unsigned byte to Register 0 Convert to float
LOADWORD	Float	F6	www		0	Write signed word to Register 0 Convert to float
LOADUWORD	Float	F7	www		0	Write unsigned word to Register 0 Convert to float
READSTR		F8		aa ... 00		Read zero terminated string from string buffer
ATOF	Float	F9	aa ... 00		0	Convert ASCII to float Store in A
FTOA	Float	FA	ff			Convert float to ASCII Store in string buffer
ATOL	Long	FB	aa ... 00		0	Convert ASCII to long Store in A
LTOA	Long	FC	ff			Convert long to ASCII Store in string buffer
FSTATUS	Float	FD		ss		Get floating point status of A
FUNCTION		FE0n				User functions 0-15
FUNCTION		FE1n				User functions 16-31
FUNCTION		FE2n				User functions 32-47
FUNCTION		FE3n				User functions 48-63
LWRITEA	Long	FEAx	yyyy zzzz			Write register and select A
LWRITEB	Long	FEBx	yyyy zzzz		0	Write register and select B
LREAD	Long	FECx		yyyy zzzz		Read register
LUDIV	Long	FEDx			0	$A = A / B$ (unsigned long)
POWER	Float	FEE0				$A = A ** B$
ROOT	Float	FEE1				$A =$ the Bth root of A
MIN	Float	FEE2				$A =$ minimum of A and B
MAX	Float	FEE3				$A =$ maximum of A and B
FRACTION	Float	FEE4			0	Load Register 0 with the fractional part of A
ASIN	Float	FEE5				$A = \text{asin}(A)$ radians
ACOS	Float	FEE6				$A = \text{acos}(A)$ radians
ATAN	Float	FEE7				$A = \text{atan}(A)$ radians
ATAN2	Float	FEE8				$A = \text{atan}(A/B)$
LCOMPARE	Long	FEE9		ss		Compare A and B (signed long integer)
LUCOMPARE	Long	FEEA		ss		Compare A and B (unsigned long integer)
LSTATUS	Long	FEEB		ss		Get long status of A
LNEGATE	Long	FEEC				$A = -A$
LABS	Long	FEED				$A = A $
LEFT		FEEE				Right parenthesis
RIGHT		FEEF			0	Left parenthesis
LOADZERO	Either	FEF0			0	Load Register 0 with zero

Opcode Name	Data Type	Opcode	Arguments	Returns	B Reg	Description
LOADONE	Float	FEF1			0	Load Register 0 with 1.0
LOADE	Float	FEF2			0	Load Register 0 with e
LOADPI	Float	FEF3			0	Load Register 0 with pi
LONGBYTE	Long	FEF4	bb		0	Write signed byte to Register 0 Convert to long
LONGUBYTE	Long	FEF5	bb		0	Write unsigned byte to Register 0 Convert to long
LONGWORD	Long	FEF6	www		0	Write signed word to Register 0 Convert to long
LONGUWORD	Long	FEF7	www		0	Write unsigned word to Register 0 Convert to long
IEEEMODE		FEF8				Set IEEE mode (default)
PICMODE		FEF9				Set PIC mode
BREAK		FEFB				Debug breakpoint
TRACEOFF		FEFC				Turn debug trace off
TRACEON		FEFD				Turn debug trace on
TRACESTR		FEFE				Send debug string to trace buffer
CHECKSUM		FEFF			0	Calculate code checksum
VERSION		FF				Copy version string to string buffer

Notes:

Data Type	data type required by opcode
Opcode	hexadecimal opcode value
Aruments	additional data required by opcode
Returns	data returned by opcode
B Reg	value of B register after opcode executes
x	register number (0-15)
n	function number (0-63)
YYYY	most significant 16 bits of 32-bit value
zzzz	least significant 16 bits of 32-bit value
ss	status byte
bb	8-bit value
www	16-bit value
aa ... 00	zero terminated ASCII string