Introduction
The uM-FPU64 IDE provides a compiler and assembler for generating uM-FPU64 code that runs on various microcontrollers, or stored as user-defined functions programmed in Flash memory on the FPU. The output format for the target microcontrollers is derived from a target description files, which specifies the correct syntax and output format. Target description files are provided for most popular microcontrollers, and others can easily be created or customized by the user.

The IDE has as a built-in editor for entering and editing source code. Symbol definitions can include constants, FPU registers, pointers, arrays, and microcontroller variables. Math equations can use long integer or floating point values, and can contain defined symbols, math operators, functions and parentheses.

The IDE compiler produces very efficient code for the FPU, and supports an extensive list of functions and procedures. Most code can be implemented using high-level compiler instructions, but a built-in assembler is also available if required.
# Table of Contents

**Introduction** ......................................................................................................................... 1
**Table of Contents** .................................................................................................................... 2
**Compiler Overview** ................................................................................................................ 7
  - Compiling ............................................................................................................................... 7
  - Comments ............................................................................................................................... 7
  - Symbol Names ........................................................................................................................ 7
  - Register Data Types .............................................................................................................. 7
  - Pre-defined Register Names .................................................................................................. 7
  - User-defined Register Names .............................................................................................. 7
  - Register Bits ......................................................................................................................... 8
  - Register Arrays .................................................................................................................... 8
  - Pointers ................................................................................................................................ 9
  - Pointer Arrays ..................................................................................................................... 9
  - Register X ............................................................................................................................. 9
  - Indirect Register .................................................................................................................. 10
  - Pointer Arithmetic ............................................................................................................... 10
  - Decimal Constants ............................................................................................................... 10
  - Hexadecimal Constants ................................................................................................------- 10
  - Floating Point Constants ..................................................................................................... 10
  - Pre-defined Constants ......................................................................................................... 10
  - User-defined Constants ....................................................................................................... 11
  - String Constants .................................................................................................................. 11
  - Microcontroller Variables .................................................................................................... 11
  - Operators ............................................................................................................................... 12
  - Operator Precedence ........................................................................................................... 12
  - Math Functions ..................................................................................................................... 13
  - User-Defined Functions and Procedures .......................................................................... 13
    - Function Prototypes ......................................................................................................... 14
    - XOP (extended opcode) Instructions ............................................................................. 14
    - Global Symbols vs Local Symbols ................................................................................. 15
    - Control Statements .......................................................................................................... 15
    - Assembler Code ................................................................................................................. 16
    - Wait Code .......................................................................................................................... 16
**Summary of Statements and Functions** .................................................................................... 17
  - Control Statements .............................................................................................................. 17
  - Function Directives ............................................................................................................. 17
  - Math Functions ..................................................................................................................... 17
  - ADC Functions ..................................................................................................................... 18
  - Serial Input/Output ............................................................................................................... 18
  - String Functions .................................................................................................................. 19
  - Timer Functions ................................................................................................................... 19
  - Matrix Functions .................................................................................................................. 19
  - Indirect Pointers .................................................................................................................. 20
  - External Input / Output ....................................................................................................... 20
  - Miscellaneous Functions .................................................................................................... 21
  - Debug Functions .................................................................................................................. 21
  - Directives ............................................................................................................................ 21
<table>
<thead>
<tr>
<th>Reference Guide: Assembler</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler Instructions</td>
<td>120</td>
</tr>
<tr>
<td>Data Directives</td>
<td>122</td>
</tr>
<tr>
<td>Symbol Definitions</td>
<td>122</td>
</tr>
<tr>
<td>Branch and Return Instructions</td>
<td>122</td>
</tr>
<tr>
<td>Condition Codes</td>
<td>123</td>
</tr>
<tr>
<td>Labels</td>
<td>123</td>
</tr>
<tr>
<td>Using Branch Instructions and Labels</td>
<td>123</td>
</tr>
</tbody>
</table>
If Statement ................................................................. 123
Repeat Statement ....................................................... 124
For Statement .......................................................... 124
String Arguments ...................................................... 125
Table Instructions .................................................... 125
MOP Instruction ....................................................... 125
Reference Guide: Target Description File ........................................ 127
Syntax ................................................................. 128
Tab Spacing ............................................................ 128
Commands ............................................................. 128
Reviewing the Sample File ......................................... 129
Reserved Words ....................................................... 131
Target Description Commands .................................... 132
  BYTE_DEFINITION ............................................. 132
  COMMENT_PREFIX .......................................... 132
  CONTINUATION .............................................. 132
  DECIMAL_FORMAT .......................................... 132
  FLOAT_DEFINITION ........................................ 133
  HEX_FORMAT .................................................. 133
  LONG_DEFINITION ........................................ 133
  MAX_LENGTH .................................................. 133
  MAX_WRITE ................................................... 133
  OPCODE_PREFIX ............................................ 134
  PRINT_FLOAT ................................................. 134
  PRINT_FPUSTRING ......................................... 134
  PRINT_LONG ............................................... 134
  PRINT_NEWLINE ........................................... 135
  PRINT_STRING ............................................. 135
  READ_BYTE .................................................. 135
  READ_CMD ................................................... 135
  READ_DELAY ................................................ 135
  READ_LONG .................................................. 136
  READ_WORD .................................................. 136
  REGISTER_DEFINITION .................................. 136
  RESERVED_PREFIX ....................................... 136
  RESERVED_WORD ......................................... 137
  SEPARATOR .................................................. 137
  SOURCE_PREFIX ........................................... 137
  START_READ_TRANSFER ................................... 137
  START_WRITE_TRANSFER .................................. 137
  STOP_TRANSFER ............................................. 138
  STRING_HEX_FORMAT ....................................... 138
  TAB_SPACING ............................................... 138
  TARGET_NAME .............................................. 138
  TARGET_OPTIONS ......................................... 139
  WAIT ......................................................... 139
  WORD_DEFINITION ......................................... 139
  WRITE ......................................................... 140
  WRITE_BYTE ................................................ 140
WRITE_BYTE_FORMAT ................................................................. 140
WRITE_CMD ................................................................. 140
WRITE_DATA ............................................................... 141
WRITE_LONG ............................................................... 141
WRITE_LONG_FORMAT ...................................................... 141
WRITE_WORD ............................................................... 142
Compiler Overview

The following section provides an overview of the compiler features.

Compiling

The compiler source code is entered into the IDE source window. The target is selected from a pop-up menu in the source window. The source code is compiled automatically when a source file is opened, or manually when the user presses the Compile button.

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:

getAddress
latitude1
NMEA_Degrees

Register Data Types

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 registers can contain any value, but the compiler requires the data type for code generation. The data types are as follows:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>32-bit or 64-bit IEEE 754 format</td>
</tr>
<tr>
<td>Long</td>
<td>32-bit or 64-bit signed integer</td>
</tr>
<tr>
<td>Unsigned</td>
<td>32-bit or 64-bit unsigned integer</td>
</tr>
</tbody>
</table>

Pre-defined Register Names

The following register names are pre-defined:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0, F1, F2, ..., F255</td>
<td>specifies a register that contains a Float data type</td>
</tr>
<tr>
<td>L0, L1, L2, ..., L255</td>
<td>specifies a register that contains a Long data type</td>
</tr>
<tr>
<td>U0, U1, U2, ..., U255</td>
<td>specifies a register that contains an Unsigned data type</td>
</tr>
</tbody>
</table>

Each register has three pre-defined names (e.g. F1, L1, and U1). All refer to the same register, but the data type is used by the compiler for code generation.

User-defined Register Names

Names can be assigned to registers with the EQU operator. The name is specified on the left side of the EQU operator. A previously assigned register name, or one of the sequential assignment symbols F%, L%, or U% is specified on the
right side of the \texttt{EQU} operator. The sequential assignment symbol assigns the name to the next register in sequence after the last register that was defined. The data type is specified by the leading character.

\begin{center}
\begin{tabular}{|l|l|}
\hline
\texttt{x} & \texttt{EQU F10} \quad \texttt{x} \text{ is assigned to register 10, data type is Float} \\
\texttt{y} & \texttt{EQU F1} \quad \texttt{y} \text{ is assigned to register 11, data type is Float} \\
\texttt{radius} & \texttt{EQU F130} \quad \texttt{radius} \text{ is assigned to register 130, data type is Float} \\
\texttt{rate} & \texttt{EQU L1} \quad \texttt{rate} \text{ is assigned to register 131, data type is Long} \\
\texttt{radius} = 1.5 & \text{Sets \texttt{radius} to the value 1.5} \\
\texttt{radius} = \texttt{radius} + 0.5 & \text{Adds the value 0.5 to \texttt{radius}} \\
\hline
\end{tabular}
\end{center}

**Register Bits**

The compiler supports bit manipulation in registers using a \textit{register.bit} notation. Where \textit{register} is the name of any FPU register, and \textit{bit} is a constant from 0 to 31 for 32-bit registers, and from 0 to 63 for 64-bit register. The register.bit notation can be used to set bits, and test bits. The notation is used as follows:

Set bit to 0:
\[
\texttt{register.bit} = 0
\]

Set bit to 1:
\[
\texttt{register.bit} = 1
\]

Toggle the bit value:
\[
\texttt{register.bit} = \neg \texttt{register.bit}
\]

Test for bit = 0:
\[
\text{if } \texttt{register.bit} = 0 \text{ then ...} \\
\text{if } \neg \texttt{register.bit} \text{ then ...} \\
\text{if } \texttt{register.bit} <> 1 \text{ then ...}
\]

Test for bit = 1:
\[
\text{if } \texttt{register.bit} = 1 \text{ then ...} \\
\text{if } \texttt{register.bit} \text{ then ...} \\
\text{if } \texttt{register.bit} <> 0 \text{ then ...}
\]

\begin{center}
\begin{verbatim}
fileMode equ L10
READ_FILE con 4
APPEND con 5

fileMode.READFILE = 1 \quad ; set bit 4 in register fileMode to 1

if fileMode.READFILE then \quad ; execute code if bit 4 in fileMode is 1
read(0)
endif
\end{verbatim}
\end{center}

**Register Arrays**

Register arrays can be assigned using the \texttt{EQU} operator. Arrays can have up to three dimensions, be defined for both 32-bit and 64-bit registers, and have a data type of Long, Unsigned or Float. The index values for arrays are specified using a constant or Long register. Register arrays with two dimensions and Float data type can be used with the matrix instructions.
Overview

Micromega Corporation

9
er

uM-FPU64 IDE - Compiler r411

### Pointers

Pointers can be defined to point to data values in registers, RAM, or Flash memory using the **PTR** operator.

The name is specified on the left side of the **PTR** operator. A previously assigned register name, or one of the sequential assignment symbols **F%, L%, or U%** is specified on the right side of the **PTR** operator. The sequential assignment symbols assigns the name to the next register in sequence after the last register that was defined. The data type is specified by the leading character. If the pointer is a 64-bit register, only the lower 32 bits are used for the pointer. The compiler uses the data type of a pointer to determine the data type of the value it points to. The value of a pointer must be assigned before it is used (see **SETIND** function). Pointers can optionally have an offset value added to them. The offset value must be previously stored in a register.

- **P1** **PTR** **F20**
  - **P1** is assigned to register 20, the data type pointed to is Float
- **P2** **PTR** **F%**
  - **P2** is assigned to register 21, the data type pointed to is Float
- **val** **EQU** **F%**
  - **val** is assigned to register 22, the data type pointed to is Float
- **n** **EQU** **L%**
  - **n** is assigned to register 23, data type is Long
- **P3** **PTR** **L130**
  - **P3** is assigned to register 130, the data type pointed to is Long
- **[p1] = val**
  - store **val** to the data location pointed at by **p1**
- **[p1+n] = val**
  - store **val** to the data location pointed at by **p1**, offset by the value in **n**
- **val = [p2]**
  - set **val** to the data value pointed at by **p2**

### Pointer Arrays

Pointers can also be used to be defined array pointers using the **PTR** operator. Arrays can be stored in registers, RAM, or Flash memory. Memory arrays can be int8, uint8, int16, uint16, long32, float32, long64, or float64 data types. Array pointers can have up to three dimensions, and can be defined for both 32-bit and 64-bit registers, with Long, Unsigned or Float data types. Array values can be used on both the left and right side of an equation. The index values for arrays are specified using a constant or Long register.

- **fp[10]** **EQU** **F10**
  - array pointer **fp** is assigned to register 10, the array data type is Float
- **mp[2,3]** **EQU** **F%**
  - array pointer **mp** is assigned to register 11, the array data type is Float
- **val** **EQU** **F%**
  - **val** is assigned to register 26, data type is Float
- **n** **EQU** **L%**
  - **n** is assigned to register 27, data type is Long
- **mp[1,2] = val**
  - assign a value to the element of the array pointed to by **mp**
- **val = fp[n]**
  - use the value of the array element pointed to by **fp**

### Register X

The register X is a pointer to another register, that auto-increments each time it is used. The **[X]** symbol is used to specify the register pointed to by register X.

- **val** **EQU** **F10**
  - **val** is assigned to register 10, data type is Float
- **fval[10]** **EQU** **F20**
  - array **fval** is assigned to registers 20 to 29, data type is Float
- **selectX(fval)**
  - set register X to point to register **fval**
- **[X] = val**
  - stores **val** to the register pointed at by register X, then increments X
- **val = [X]**
  - set **val** to the value of the register pointed at by register X, then increments X
**Indirect Register**
If a register is enclosed in square brackets, the lower 8 bits of the register value are used to reference another register.

\[
\begin{array}{c|c}
m & \text{EQU L10} \\
\end{array}
\]
\(m\) is assigned to register 10, data type is Long

\[
\begin{array}{c}
[m] = \text{val} \\
\end{array}
\]
stores value to register pointed at by the lower 8 bits of \(m\)

**Pointer Arithmetic**
Several compiler statements are provided for pointers. Pointers can be set using the `SETIND` function, incremented, decremented, and the size in bytes between two pointer can be calculated (the pointers must both have the same data type).

\[
\begin{array}{c|c}
P1 & \text{PTR F20} \\
P2 & \text{PTR F}% \\
n & \text{EQU L}% \\
\end{array}
\]

\(P1\) is assigned to register 20, the data type pointed to is Float

\(P2\) is assigned to register 21, the data type pointed to is Float

\(n\) is assigned to register 23, data type is Long

\[
\begin{array}{c}
p1 = \text{SETIND(REG_FLOAT, F40)} \\
p2 = p1 \\
p1 = p1 + 1 \\
p1 = p1 - 1 \\
n = p2 - p1 \\
\end{array}
\]
set \(p1\) to point to register 40, data type is register, Float
set \(p2\) to the value of \(p1\)
increment \(p1\)
decrement \(p1\)
set \(n\) to the number of bytes between \(p1\) and \(p2\)

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

\[
\begin{array}{c}
120 \\
-53 \\
100000 \\
+207 \\
\end{array}
\]

**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.

\[
\begin{array}{c}
$55 \\
0xFF \\
$FFFF \\
0x13 \\
\end{array}
\]

**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.

\[
\begin{array}{c}
1.0 \\
-53 \\
1E6 \\
-1.5e-3 \\
\end{array}
\]

**Pre-defined Constants**

\[
\begin{array}{c}
\text{PI} \quad \text{constant value for pi (32-bit: 3.1415926, 64-bit: 3.141592653589793)} \\
\text{E} \quad \text{constant value for e (32-bit: 2.7182818, 64-bit: 2.718281828459045)} \\
\end{array}
\]
**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 prefix. The special characters are as follows:

- \r carriage return (0x0D)
- \n linefeed (0x0A)
- \t horizontal tab (0x09)
- \v vertical tab (0x0B)
- \ backslash
- \" double quote
- \xx 8-bit value (where xx are hexadecimal digits, e.g. \0C"

<table>
<thead>
<tr>
<th>String Constant</th>
<th>Actual String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;sample&quot;</td>
<td>sample</td>
</tr>
<tr>
<td>&quot;string2\r\n&quot;</td>
<td>string2&lt;carriage return&gt;&lt;linefeed&gt;</td>
</tr>
<tr>
<td>&quot;\0C\FF&quot;</td>
<td>binary values: 0C, FF</td>
</tr>
<tr>
<td>&quot;5\3&quot;</td>
<td>5\3</td>
</tr>
<tr>
<td>&quot;this &quot;one&quot;&quot;</td>
<td>this &quot;one&quot;</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>count</th>
<th>EQU</th>
<th>BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sensorInput</td>
<td>EQU</td>
<td>UWORD</td>
</tr>
<tr>
<td>lastAngle</td>
<td>EQU</td>
<td>FLOAT</td>
</tr>
</tbody>
</table>

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.

```
<table>
<thead>
<tr>
<th>degreesC</th>
<th>EQU</th>
<th>BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>degreesF</td>
<td>EQU</td>
<td>F10</td>
</tr>
</tbody>
</table>
```
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 |

**Operators**
The following operators are supported by the compiler.

+  Addition  
-  Subtraction  
*  Multiplication  
/  Division  
%  Modulo  
**  Power  
|  Bitwise-OR  
^  Bitwise-XOR  
&  Bitwise-AND  
<<  Shift left  
>>  Shift right  
~  Ones complement  
+  Unary plus  
-  Unary minus

**Operator Precedence**
Math equations are evaluated by the IDE using the following operator precedence.

\[ \sim + - * / \% ** + - << >> & ^ | \]

\[ F1 = F2 + F3 \times F4 \]
results in \( F1 \) being set to the value of \( F3 \) multiplied by \( F4 \), then added to \( F2 \). Parenthesis can be used to change the order of evaluation.

\[ F1 = (F2 + F3) \times F4 \]
results in \( F1 \) being set to the value of \( F2 \) added to \( F3 \) then multiplied by \( F4 \). Multiple constant values entered one after another are automatically reduced to a single constant in the expression.

\[ F1 = F2 \times 5 \div 2 \]
results in \( F_1 \) being set to the value \( F_2 \) multiplied by 2.5. If you don’t want constants to be reduced, you need to use parentheses.

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.

## 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 \texttt{FABS, FMOD, FMIN, FMAX} instructions for floating point data types, and the \texttt{LABS, LDIV} (remainder), \texttt{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 \textit{uM-FPU64 Instruction Set} document.

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

\[
\text{theta} = \sin(\text{angle})
\]

\[
\text{fcube} = \text{power}(f, 3)
\]

\[
\text{result} = \cos(\text{PI}/2 + \sin(\text{theta}))
\]
User-defined functions and procedures are defined 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`, `#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 and procedure calls can be nested up to 16 levels deep.

Procedures are functions with no return value. They can have up to nine parameters. Procedures are called from a separate source line, and can’t be used in expressions.

```
#function 1 initDevice
...  
#end

initDevice
```

```
#function 1 convert(float) float  
  return arg1 * 9/5 + 32  
#end

tempF = convert(tempC)
```

Functions have return values. They can have up to nine parameters. Functions are called from within an expression.

```
#function 1 convert(float) float  
  return arg1 * 9/5 + 32  
#end

tempF = convert(tempC)
```

**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  
GetCircumference func   %
GetArea func   %
```

**XOP (extended opcode) Instructions**

XOP (extended opcode) instructions can be loaded from XOP library files and stored in Flash memory. For example a set of quaternion instructions are defined in the quaternion library file. Each XOP used in a program must have an `#XOP` directive specified previously in the program to load the XOP code and define the arguments and return value (if any).

```
#xop quaternion:q_add  
#xop quaternion:q_norm  

q_add(qa, qb, qc)
    qa = qb + qc, add quaternions

tmp = q_norm(qa)
    calculate the norm of quaternion qa
```
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.

```assembly
    tmp1 equ F1
    #function sample1
    tmp2 equ F2
    SELECTA, tmp1
    FSET, tmp2
    #end
```

global symbol definition

local symbol definition

both tmp1 and tmp2 are defined inside the function

only tmp1 is defined outside the function

Control Statements
The following control statements are supported by the compiler.

CONTINUE

DO | [DO] WHILE condition1
    statements
    [CONTINUE]
    [EXIT]

LOOP | [LOOP] UNTIL condition2

EXIT

FOR register = startExpression TO | DOWNTO endExpression [STEP stepExpression]
    [statements]
    [CONTINUE]
    [EXIT]

NEXT

IF condition THEN CONTINUE
IF condition THEN EXIT
IF condition THEN RETURN
IF condition THEN equalsStatement

IF condition THEN
    statements
    [ELSEIF condition THEN
        statements]...
    [ELSE
        statements]
ENDIF

RETURN [returnValue]

SELECT compareItem

[CASE compareValue [, compareValue]...
    statements]...
[ELSE
    statements]...
Overview

**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 section entitled *Reference Guide: Assembler* for more details on assembler code.

```
#ASM
  SELECTA, 1
  LOADPI
  FSET0
  FDIVI, 2
#ENDASM
```

**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.
Summary of Statements and Functions

Control Statements

CONTINUE

DO | [DO] WHILE condition1
    statements
    [CONTINUE]
    [EXIT]
LOOP | [LOOP] UNTIL condition2

EXIT

FOR register = startExpression TO | DOWNTO endExpression [STEP stepExpression]
    [statements]
    [CONTINUE]
    [EXIT]
NEXT

IF condition THEN CONTINUE
IF condition THEN EXIT
IF condition THEN RETURN
IF condition THEN equalsStatement

IF condition THEN
    statements
[ELSEIF condition THEN
    statements]...
[ELSE
    statements]
ENDIF

RETURN [returnValue]

SELECT compareItem
    statements
[CASE compareValue [, compareValue]...
    statements]...
[ELSE
    statements]
ENDESELECT

STATUS(conditionCode)

Function Directives

#FUNC number name([arg1Type, arg2Type, ...])
#FUNC number name([arg1Type, arg2Type, ...]) returnType
#FUNCTION number name([arg1Type, arg2Type, ...])
#FUNCTION number name([arg1Type, arg2Type, ...]) returnType

Math Functions

result = SQRT(arg1)
result = LOG(arg1)
result = LOG10(arg1)
result = EXP(arg1)
result = EXP10(arg1)
result = SIN(arg1)
result = COS(arg1)
result = TAN(arg1)
result = ASIN(arg1)
result = ACOS(arg1)
result = ATAN(arg1)
result = ATAN2(arg1, arg2)
result = DEGREES(arg1)
result = RADIANS(arg1)
result = FLOOR(arg1)
result = CEIL(arg1)
result = ROUND(arg1)
result = POWER(arg1, arg2)
result = ROOT(arg1, arg2)
result = FRAC(arg1)
result = INV(arg1)
result = FLOAT(arg1)
result = FIX(arg1)
result = FIXR(arg1)
result = ABS(arg1)
result = MOD(arg1, arg2)
result = MIN(arg1, arg2)
result = MAX(arg1, arg2)

ADC Functions
result = ADCFLOAT(channel)
result = ADCLONG(channel)
ADCMODE(MANUAL_TRIGGER, repeat)
ADCMODE(EXTERNAL_TRIGGER, repeat)
ADCMODE(TIMER_TRIGGER, repeat, period)
ADCMODE(DISABLE)
ADCScale(channel, scaleFactor)
ADCTRIG
ADCMWAIT

Serial Input/Output
SERIAL(SET_BAUD, baud)
SERIAL(WRITE_TEXT, string)
SERIAL(WRITE_TEXT2, string)
SERIAL(WRITE_STRBUF)
SERIAL(WRITE_STRSEL)
SERIAL(WRITE_CHAR, value)
SERIAL(WRITE_FLOAT, value, format)
SERIAL(WRITE_LONG, value, format)
SERIAL(WRITE_COMMA)
SERIAL(WRITE_CRLF)
**Serial Functions**

- SERIAL(DISABLE_INPUT)
- SERIAL(ENABLE_CHAR)
- SERIAL STATUS_CHAR
- result = SERIAL(READ_CHAR)
- SERIAL(ENABLE_NMEA)
- SERIAL STATUS_NMEA
- SERIAL(READ_NMEA)

**String Functions**

- FTOA(value, format)
- LTOA(value, format)
- STRBYTE(value)
- STRFCHR(string)
- STRFIELD([field])
- STRFIND(string)
- result = STRFLOAT()
- STRING(increment)
- STRINS(string)
- result = STRLONG()
- STRSEL([start], length)
- STRSET(string)

**Timer Functions**

- result = TICKLONG()
- result = TIMELONG()
- TIMESET(seconds)
- DELAY(period)

**Matrix Functions**

- FFT(type)
- result = LOADMA(row, column)
- result = LOADMB(row, column)
- result = LOADMC(row, column)
- MOP(SCALAR_SET, value)
- MOP(SCALAR_ADD, value)
- MOP(SCALAR_SUB, value)
- MOP(SCALAR_SUBR, value)
- MOP(SCALAR_MUL, value)
- MOP(SCALAR_DIV, value)
- MOP(SCALAR_DIVR, value)
- MOP(SCALAR_POW, value)
- MOP(EWISE_SET)
- MOP(EWISE_ADD)
- MOP(EWISE_SUB)
- MOP(EWISE_SUBR)
- MOP(EWISE_MUL)
- MOP(EWISE_DIV)
- MOP(EWISE_DIVR)
- MOP(EWISE_POW)
- MOP(MULTIPLY)
- MOP(IDENTITY)
- MOP(DIAGONAL, value)
- MOP(TRANSPOSE)
- return = MOP(COUNT)
return = MOP(SUM)
return = MOP(AVE)
return = MOP(MIN)
return = MOP(MAX)
MOP(COPYAB)
MOP(COPYAC)
MOP(COPYBA)
MOP(COPYBC)
MOP(COPYCA)
MOP(COPYCB)
return = MOP(DETERMINANT)
MOP(LOADRA)
MOP(LOADRB)
MOP(LOADRC)
MOP(LOADBA)
MOP(LOADCA)
MOP(SAVEAR)
MOP(SAVEAB)
MOP(SAVEAC)
SAVEMA(row, column, value)
SAVEMB(row, column, value)
SAVEMC(row, column, value)
SELECTMA(register|pointer, rows, columns)
SELECTMB(register|pointer, rows, columns)
SELECTMC(register|pointer, rows, columns)

Indirect Pointers
COPYIND(fromPointer, toPointer, count)
SETIND(type, register)
SETIND(type, address)
SETIND(type, function, offset)

External Input / Output
DEVIO(device, action [, ...])
return = DEVIO(device, action [, ...])
DIGIO(pin+action [, ...])
return = DIGIO(pin+action [, ...])
result = EXTLONG()
EXTSET(value)
EXTWAIT
Miscellaneous Functions

EVENT(action [, ...])
result = FCNV(value, conversion)
result = FLOOKUP(value, item0, item1, ...)
result = FTABLE(value, cc, item0, item1, ...)
result = LLOOKUP(value, item0, item1, ...)
result = LTABLE(value, cc, item0, item1, ...)
result = POLY(value, coeff1, coeff2, ...)
result = READVAR(number)
RTC(action [, ...])
result = RTC(action)
SELECTA(register)
SELECTX(register)

Debug Functions

BREAK
TRACEON
TRACEOFF
TRACEREG(register)
TRACESTR(string)

Directives

#ASM
#END
#ENDASM
#FIRMWARE_REQUIRED number
#FUNC number name[arg1Type, arg2Type, ...]
#FUNC number name[arg1Type, arg2Type, ...] returnType
#FUNCTION number name[arg1Type, arg2Type, ...]
#FUNCTION number name[arg1Type, arg2Type, ...] returnType
#IDE_REQUIRED number
#TARGET_OPTIONS target, ...
Reference Guide: Compiler

ADCFLOAT

Returns the scaled floating point value from the last reading of the specified ADC channel.

Syntax

\[ \text{result} = \text{ADCFLOAT}(\text{channel}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>float</td>
<td>The last ADC reading from the selected channel, multiplied by</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the scale factor.</td>
</tr>
<tr>
<td>channel</td>
<td>long constant</td>
<td>ADC channel. (0-8)</td>
</tr>
</tbody>
</table>

Notes

This function waits until the Analog-to-Digital conversion is complete, then returns the floating point value from the last reading of the specified ADC channel, multiplied by the scale factor specified for that channel. The scale factor is set by the ADCSCALE procedure (the default scale factor is 1.0). This function will only wait if the instruction buffer is empty. If there are other instructions in the instruction buffer, or another instruction is sent before the ADCFLOAT function has been completed, the function will terminate and the previous value for the selected channel will be returned.

Examples

\[
\text{result} = \text{ADCFLOAT}(0) \\
\quad ; \text{returns the value for A/D channel 0} \\
\quad ; \text{if A/D reading is 200, and scale multiplier} = 1.0, \text{result} = 200.0 \\
\quad ; \text{if A/D reading is 200, and scale multiplier} = 1.5, \text{result} = 300.0
\]

See Also

ADCLONG, ADCMODE, ADCSCALE, ADCTRIG, ADCWAIT

\[ uM-FPU64 \text{ Instruction Set: ADCLOAD } \]
ADCLONG

Returns the long integer value from the last reading of the specified ADC channel.

Syntax

\[ \text{result} = \text{ADCLONG}(\text{channel}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The last ADC reading from the selected channel.</td>
</tr>
<tr>
<td>channel</td>
<td>long constant</td>
<td>A/D channel. (0 or 1)</td>
</tr>
</tbody>
</table>

Notes

This function waits until the Analog-to-Digital conversion is complete, then returns the long integer value from the last reading of the specified ADC channel. This function will only wait if the instruction buffer is empty. If there are other instructions in the instruction buffer, or another instruction is sent before the ADCLONG function has been completed, the function will terminate and the previous value for the selected channel will be returned.

Examples

\[ \text{result} = \text{ADCLONG}(0) \] ; returns the value for A/D channel 0
\[ \text{; if A/D channel 0 is 200, result = 200} \]

See Also

ADCFLOAT, ADCMODE, ADCSCALE, ADCTRIG, ADCWAIT

uM-FPU64 Instruction Set: ADCLONG
ADCMODE

Set the trigger mode of the Analog-to-Digital Converter (ADC).

Syntax

```
ADCMODE(MANUAL_TRIGGER, repeat)
ADCMODE(EXTERNAL_TRIGGER, repeat)
ADCMODE(TIMER_TRIGGER, repeat, period)
ADCMODE(DISABLE)
```

Notes

When the ADC is triggered the ADC channels are sampled, and the repeat count specifies the number of additional samples that are taken. The ADC reading is the average of all samples. There are three ADC trigger modes: Manual, External, and Timer.

When the ADC is enabled for manual trigger, the Analog-to-Digital conversions are triggered by calling the ADCTRIG procedure. If a conversion is already in progress, the trigger is ignored. This mode is the easiest to use since the trigger is software controlled. Manual trigger is used for applications that only require occasional Analog-to-Digital sampling, or that don’t require a periodic sampling rate.

When the ADC is configured for external trigger, Analog-to-Digital conversions are triggered by the rising edge of the input signal on the EXTIN pin. To avoid missing samples, the program must read the ADC value before the next trigger occurs. External input trigger is used for applications that need to synchronize that Analog-to-Digital conversion with an external signal.

When the ADC is configured for timer trigger, Analog-to-Digital conversions are triggered at a specific time interval. The time interval is set with the period parameter, which specifies the time interval in microseconds. The minimum time interval is 100 microseconds and the maximum time interval is 4294.967 seconds. Short time intervals (from 100 microseconds to 2 milliseconds) are accurate to the microsecond, whereas longer time intervals (greater than 2 milliseconds) are accurate to the millisecond. To avoid missing samples, the program must read the ADC value before the next trigger occurs. Timer trigger is used for applications that need to sample an analog input at a specific frequency.

The ADC can be disabled by calling the ADCMODE(DISABLE) procedure.

Examples

```
ADCMODE(MANUAL_TRIGGER, 0) ; manual trigger, 1 sample per trigger
ADCMODE(EXTERNAL_TRIGGER, 4) ; external input trigger, 5 samples per trigger
ADCMODE(TIMER_TRIGGER, 0, 1000) ; timer trigger every 1000 usec, 1 sample per trigger
```

See Also

ADCFLOAT, ADCLONG, ADCSCALE, ADCTRIG, ADCWAIT

*References*

uM-FPU64 Instruction Set: ADCMODE
ADCScale

Sets the scale value for the ADC channel.

Syntax

```
ADCScale(channel, scaleFactor)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>channel</td>
<td>long constant</td>
<td>ADC channel (0 or 1).</td>
</tr>
<tr>
<td>scaleFactor</td>
<td>float expression</td>
<td>Scale factor.</td>
</tr>
</tbody>
</table>

Notes

This sets the scale value for the specified ADC channel. The scale factor is used by the ADCFLOAT instruction to return a scaled, floating point ADC value.

Examples

The following example scales the ADC readings so that ADCFLOAT returns the analog value in volts. The scale factor is set to the operating voltage (3.3V), divided by the the number of ADC steps (the uM-FPU64 FPU has a 12-bit ADC, so there are 4095 steps).

```
ADCScale(0, 3.3/4095) ; set scale factor for channel 0 for range of 0.0 to 3.3V
```

See Also

ADCFLOAT, ADCLONG, ADCMODE, ADCTRIG, ADCWAIT

uM-FPU64 Instruction Set: ADCSCALE
ADCTRIG

Triggers an ADC conversion.

Syntax

```
ADCTRIG
```

Notes

This procedure is only required if the ADC trigger mode has been set to manual.

Examples

```
; setup
ADCMODE(MANUAL_TRIGGER, 0) ; set manual trigger, 1 sample per trigger

; sample
ADCTRIG ; trigger the conversion
adcVal = ADCFLOAT(0) ; get the ADC value from channel 0
```

See Also

ADCFLOAT, ADCLONG, ADCMODE, ADCSCALE, ADCTRIG, ADCWAIT

uM-FPU64 Instruction Set: ADCTRIG
**ADCWAIT**

Waits until the next ADC value is ready.

**Syntax**

```
ADCWAIT
```

**Notes**

This procedure is used to wait until the next ADC value is ready. This procedure only waits if the instruction buffer is empty. The IDE compiler automatically adds an FPU wait call if the procedure is called from microcontroller code. If this procedure is used in a user-defined function, the user must be sure that an FPU wait call is inserted in the microcontroller code immediately after the function call. If there are other instructions in the instruction buffer, or another instruction is sent before the ADCWAIT procedure has completed, it will terminate and return.

**Examples**

```plaintext
; setup
ADCMODE(TIMER_TRIGGER, 0, 1000) ; set timer trigger every 1000 usec, 1 sample per trigger

; sample
do
  ADCWAIT ; wait for the next ADC value
  adcVal = ADCFLOAT(0) ; get the ADC value from channel 0
loop
```

**See Also**

ADCFLOAT, ADCLONG, ADCMODE, ADCSCALE, ADCTRIG

_uM-FPU64 Instruction Set: ADCWAIT_
**BREAK**

Debug breakpoint.

**Syntax**

```
BREAK
```

**Notes**

If the debugger is enabled, a debug breakpoint occurs, and the debugger is entered. If the debugger is disabled, this procedure is ignored.

**Examples**

```
BREAK ; stop execution and enter the debugger
```

**See Also**

- TRACEOFF, TRACEON, TRACEREG, TRACESTR
- `uM-FPU64 Instruction Set: BREAK`
Conditional Expressions

Conditional expressions are used by control statements to determine if a statement or group of statements will be executed.

Syntax

```
classical expression:
  [NOT] relational expression [[AND | OR] [NOT] relational expression]...

relational expression:
  register
  register.bit
  expression
  expression < | <= | = | <> | > | >= expression
  STRSEL([[start, ]length]]) < | <= | = | <> | > | >= string constant
  STRFIELD([field]) < | <= | = | <> | > | >= string constant
  STATUS(condition code)
```

Examples

```
x equ F10
n equ L11

if n.5 then return
if log(x) < 0.3 then n = n + 1
if n then exit
if n > 1 AND n < 5 then x = 0
if NOT (n > 1 AND n < 10) or n = 5 then continue
if strfield(1) = "GPRMC" then
  ; statements
endif
if status(GT) then return
```

See Also

Expressions, DO...WHILE...UNTIL...LOOP, IF...THEN, IF...THEN...ELSE
CONTINUE

Continues execution at the next iteration of the loop.

Note: Must be used inside a FOR...NEXT or DO...WHILE...LOOP...UNTIL control statement.

Syntax

CONTINUE

Notes

Continues execution at the next iteration of the innermost loop that the CONTINUE statement is contained in.

Examples

```
n equ L10
x equ F11

FOR n = 1 TO 100
   ; statements
   if x > 1500 then CONTINUE ; continue execution at next iteration of the DO loop
   ; statements
NEXT
```

See Also

DO...WHILE...UNTIL...LOOP, EXIT, FOR...NEXT, IF...THEN, RETURN
COPYIND

Copies data values specified from the location specified by one pointer to the location specified by another pointer.

Syntax

COPYIND(fromPointer, toPointer, count)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fromPointer</td>
<td>pointer</td>
<td>From pointer.</td>
</tr>
<tr>
<td>toPointer</td>
<td>pointer</td>
<td>To pointer.</td>
</tr>
<tr>
<td>count</td>
<td>long expression</td>
<td>The number of data items to copy.</td>
</tr>
</tbody>
</table>

Notes

If the data types of the two pointers are different, the data is converted from the data type of the fromPointer to the data type of the toPointer, as the data is being copied.

Examples

```
p1 = SETIND(REG_FLOAT, F10) ; sets pointer to register 10, data type is Float
p2 = SETIND(MEM_INT8, 100) ; sets pointer to RAM address 100, data type is int8
COPYIND(p1, p2, 10) ; copies 10 data values from register 10 to 19, ; converts them the data values to integer, ; stores the lower 8 bits to RAM at address 100 to 109
```

See Also

SETIND, LOADIND, SAVEIND

uM-FPU64 Instruction Set: COPYIND
**DELAY**

Delay for the number of milliseconds.

**Syntax**

```plaintext
DELAY(period)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>period</td>
<td>long expression</td>
<td>The delay period in milliseconds.</td>
</tr>
</tbody>
</table>

**Examples**

```plaintext
DELAY(1000) ; delay for one second
```

**See Also**

TIMESET, TICKLONG, TIMELONG

*uM-FPU64 Instruction Set: DELAY*
DEVIO

Device Input/Output.

Syntax

**Actions Supported by all Devices**

```
DEVIO(device, DISABLE)
DEVIO(device, ENABLE, pin, config)
DEVIO(device, [device specific])
DEVIO(device, WRITE_REG8[+MSB][+LSB], register)
DEVIO(device, WRITE_REG16[+MSB][+LSB], register)
DEVIO(device, WRITE_REG32[+MSB]+LSB], register)
DEVIO(device, WRITE_REG64[+MSB][+LSB], register)
DEVIO(device, WRITE_BYTE, byte)
DEVIO(device, WRITE_WORD, byte, byte)
DEVIO(device, WRITE_NBYTE, count, byte, ...)
DEVIO(device, WRITE_REP, count, byte)
DEVIO(device, WRITE_STR, string)
DEVIO, device, WRITE_SBUF)
DEVIO(device, WRITE_SSEL)
DEVIO(device, WRITE_MEM, count)
DEVIO(device, WRITE_MEMA, address, count)
DEVIO(device, WRITE_MEMR, regAddr, regCount)
DEVIO(device, READ_REG8[+MSB][+LSB][+ZE][+SE], register)
DEVIO(device, READ_REG16[+MSB][+LSB][+ZE][+SE], register)
DEVIO(device, READ_REG32[+MSB][+LSB][+ZE][+SE], register)
DEVIO(device, READ_REG64[+MSB][+LSB][+ZE][+SE], register)
DEVIO(device, READ_SKIP, count)
DEVIO(device, READ_SBUF)
DEVIO(device, READ_SSEL)
DEVIO(device, READ_MEM, count)
DEVIO(device, READ_MEMA, address, count)
DEVIO(device, READ_MEMR, regAddr, regCount)
```

**Actions Supported by Loadable Devices**

```
DEVIO(device, LOAD_DEVICE, xopdev)
```

**Device Specific Actions**

```
DEVIO(COUNTER+n, DEBOUNCE, period)
DEVIO(COUNTER+n, REPEAT, delay, rate)
result = DEVIO(COUNTER+n, READ_COUNT)
result = DEVIO(COUNTER+n, EDGE1_MSEC)
result = DEVIO(COUNTER+n, EDGE1_USEC)
result = DEVIO(COUNTER+n, EDGE2_MSEC)
result = DEVIO(COUNTER+n, EDGE2_USEC)
```

```
DEVIO(FIFOn, CLEAR)
result = DEVIO(FIFOn, USED)
result = DEVIO(FIFOn, FREE)
result = DEVIO(FIFOn, STATUS)
DEVIO(FIFOn, CLEAR_OVERFLOW)
DEVIO(FIF0, CLEAR)
DEVIO(FIFOn, ALLOC_MEM, size)
DEVIO(FIFOn, ALLOC_MEMR, regSize)
```

```
DEVIO(I2C+n, START_WRITE)
DEVIO(I2C+n, STOP)
DEVIO(I2C+n, START_READ, byteCount)
```
DEVIO(LCD, CLEAR)
DEVIO(LCD, HOME)
DEVIO(LCD, MOVE, row, column)
DEVIO(LCD, MOVE_REG, rowRegister, columnRegister)
DEVIO(LCD, CMD, command)
DEVIO(LCD, INTERFACE, type)
DEVIO(LCD, BACKLIGHT_ON)
DEVIO(LCD, BACKLIGHT_OFF)

DEVIO(MEM, ALLOCATE, memSize, fifoSize)
DEVIO(MEM, ALLOCATE, memSize, fgSize)

DEVIO(OWIRE, RESET)
DEVIO(OWIRE, SELECT, addressRegister)
DEVIO(OWIRE, VERIFY, addressRegister)
DEVIO(OWIRE, SEARCH, count, addressRegister)
DEVIO(OWIRE, ALARM, count, addressRegister)
DEVIO(OWIRE, FAMILY_SEARCH, count, addressRegister)
DEVIO(OWIRE, FAMILY_ALARM, count, addressRegister)

DEVIO(SERVO+n, PULSE, register)
DEVIO(SERVO+n, SPEED, register)
DEVIO(SERVO+n, TIME, register)
DEVIO(SERVO+n, MOVE, register)
DEVIO(SERVO+n, HOME, register)

result = DEVIO(SERVO+n, READ_PULSE)
result = DEVIO(SERVO+n, STATUS)

DEVIO(SPI+n, CS_LOW)
DEVIO(SPI+n, CS_HIGH)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>pre-defined symbol and byte constant</td>
<td>The device type and number. FIFO1,FIFO2,FIFO3,FIFO4,OWIRE,I2C,SPI,ASYNC,COUNTER,SERVO,LCD,VDRIVE2</td>
</tr>
<tr>
<td>action</td>
<td>pre-defined symbols</td>
<td>The device action and modifiers.</td>
</tr>
</tbody>
</table>

**Notes**
See the *uM-FPU64 Instruction Set* document for detailed descriptions of the general DEVIO actions, and the device-specific actions.

**Examples**

DEVIO(ASYNC, ENABLE, D1, RX+BAUD_9600) ; enable pin D1 for receive at 9600 baud

**See Also**
*uM-FPU64 Instruction Set*: DEVIO
DIGIO

Set the OUT0 or OUT1 output pin.

Syntax

\[
\text{DIGIO}(pin+\text{LOW}) \\
\text{DIGIO}(pin+\text{HIGH}) \\
\text{DIGIO}(pin+\text{TOGGLE}) \\
\text{DIGIO}(pin+\text{INPUT}) \\
\text{DIGIO}(pin+\text{WRITE}_{\text{BITS}}, \text{bitCount}[+\text{PRE}][+\text{POST}][+\text{MSB}][+\text{LSB}][+\text{FAST}][+\text{SLOW}], \text{value}) \\
\text{result} = \text{DIGIO}(pin+\text{READ}_{\text{BITS}}, \text{bitCount}[+\text{PRE}][+\text{POST}][+\text{MSB}][+\text{LSB}][+\text{FAST}][+\text{SLOW}]) \\
\text{DIGIO}(pin+\text{WRITE}_{\text{BITP}}, \text{bitCount}[+\text{INVERT}], \text{bitValue}) \\
\text{result} = \text{DIGIO}(pin+\text{READ}_{\text{BITP}}, \text{bitCount}[+\text{INVERT}])
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pin</td>
<td>long constant</td>
<td>Output pin (D0 to D22).</td>
</tr>
<tr>
<td>action</td>
<td>pre-defined symbols</td>
<td>The device action and modifiers.</td>
</tr>
<tr>
<td>bitCount</td>
<td>long constant</td>
<td>The number of bits to transfer.</td>
</tr>
<tr>
<td>bitValues</td>
<td>long expression</td>
<td>The lower bits of the expression are written to pins. If no expression is included the current value in register 0 will be used. The number of bits written is determined by bitCount.</td>
</tr>
<tr>
<td>result</td>
<td>long</td>
<td>Value read from pins.</td>
</tr>
</tbody>
</table>

Notes

See the uM-FPU64 Instruction Set document for detailed descriptions of the DIGIO actions.

Examples

DIGIO(D0+LOW) ; set pin D0 to low
DIGIO(D1+TOGGLE) ; toggle the value of pin D1
DIGIO(D2+INPUT) ; set status flag according to the value of pin D2

See Also

uM-FPU64 Instruction Set: DIGIO
DO...WHILE...UNTIL...LOOP

Repeatedly execute a group of statements while specified conditions are true.

*Note:* Must be used inside a user-defined procedure or function.

**Syntax**

```
DO | [DO] WHILE condition1
    statements
    [CONTINUE]
    [EXIT]
LOOP | [LOOP] UNTIL condition2
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition1</td>
<td>While this condition is true, execute the statements in the loop.</td>
</tr>
<tr>
<td>statements</td>
<td>One or more statements to be executed each time through the loop.</td>
</tr>
<tr>
<td>condition2</td>
<td>While this condition is false, repeat the loop.</td>
</tr>
</tbody>
</table>

**Notes**

The DO loop will repeatedly execute the statements in the loop. If the WHILE clause is specified, the DO loop will terminate if `condition1` is false. If the UNTIL clause is specified, the DO loop will terminate if `condition2` is true. The WHILE clause is checked at the start of the DO loop, and the UNTIL clause is checked at the end of the DO loop. If neither a WHILE clause or UNTIL clause is specified, the DO loop will be an infinite loop, and can only be terminated by an EXIT or RETURN statement. The CONTINUE statement is used to skip ahead to the end of the DO loop. The EXIT statement is used to immediately terminate the DO loop. The RETURN statement is used to exit the user-defined function.

**Examples**

```
DO
    ; statements executed each loop iteration
LOOP

WHILE n > 0
    ; statements executed each loop iteration
LOOP

DO
    ; statements executed each loop iteration
UNTIL n > 0

DO WHILE n >= 10
    ; statements executed each loop iteration
LOOP UNTIL n > 20
```
See Also

CONTINUE, EXIT, FOR...NEXT, IF...THEN, IF...THEN...ELSE, RETURN,
SELECT...CASE
EVENT

Manage background events.

Syntax

EVENT(DISABLE+event)
EVENT(ENABLE+event, function)
EVENT(PERIOD+event[, timePeriod])
EVENT(SET+event)
EVENT(CLEAR+event)
EVENT(WAIT+event)
EVENT(TEST+event)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>pre-defined symbols</td>
<td>The event action.</td>
</tr>
<tr>
<td>event</td>
<td>pre-defined symbols</td>
<td>The device action and event.</td>
</tr>
<tr>
<td>function</td>
<td>function</td>
<td>Background function to execute when event occurs.</td>
</tr>
<tr>
<td>timePeriod</td>
<td>long expression</td>
<td>The time period in milliseconds. If no expression is included the current value in register 0 will be used.</td>
</tr>
</tbody>
</table>

Notes

See the uM-FPU64 Instruction Set document for detailed descriptions of the EVENT actions.

Examples

EVENT(ENABLE+RTC) ; enable RTC event

See Also

uM-FPU64 Instruction Set: EVENT
EXIT

Terminates the loop.

Note: Must be used inside a FOR...NEXT or DO...WHILE...LOOP...UNTIL control statement.

Syntax

EXIT

Notes

Terminates execution of the innermost loop that the EXIT statement is contained in.

Examples

```plaintext
n equ L10
x equ F11

FOR n = 1 TO 100
    ; statements
    if x > 1500 then EXIT ; exit the FOR loop
    ; statements
NEXT
```

See Also

CONTINUE, DO...WHILE...UNTIL...LOOP, EXIT, FOR...NEXT, IF...THEN, RETURN
Expressions

A primary expression consists of a register, variable, math function, or user-defined function. Primary expressions can also be combined with math operators and parenthesis to implement more complex numeric expressions.

The math operators are as follows:

<table>
<thead>
<tr>
<th>Math Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bitwise-OR</td>
</tr>
<tr>
<td></td>
<td>Bitwise-XOR</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise-AND</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift right</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulo operation</td>
</tr>
<tr>
<td>**</td>
<td>Power</td>
</tr>
<tr>
<td>~</td>
<td>Ones complement</td>
</tr>
<tr>
<td>+</td>
<td>Unary plus</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus</td>
</tr>
</tbody>
</table>

Examples

angle = \sin(n + \pi/2)
angle = (n << 8) + m \% 5
n = n ** 3

See Also

Conditional Expressions, FOR...NEXT, SELECT...CASE
**EXTLONG**

Returns the value of the external input counter.

**Syntax**

\[ \text{result} = \text{EXTLONG}() \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The value of the external input counter.</td>
</tr>
</tbody>
</table>

**Examples**

\[ \text{result} = \text{EXTLONG}() ; \text{returns the value from the external input counter} \]

**See Also**

EXTSET, EXTWAIT

*um-FPU64 Instruction Set: EXTLONG*
EXTSET

Sets the value of the external input counter.

Syntax

\texttt{EXTSET(value)}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>long expression</td>
<td>The external input counter is set to this value.</td>
</tr>
</tbody>
</table>

Notes

If \texttt{value <> -1}, the external input counter is set to that value and the counter is enabled.
If \texttt{value = -1}, the external counter is disabled.
The external counter counts the rising edges that occur on the \texttt{EXTIN} pin.

Examples

\texttt{EXTSET(0)} ; the external input counter is set to zero

See Also

\texttt{EXTLONG, EXTWAIT}

\textit{uM-FPU64 Instruction Set: EXTSET}
**EXTWAIT**

Wait for the next external input to occur.

**Syntax**

```
EXTWAIT
```

**Notes**

This procedure is used to wait until the next external input occurs. This procedure only waits if the instruction buffer is empty. The IDE compiler automatically adds an FPU wait call if the procedure is called from microcontroller code. If this procedure is used in a user-defined function, the user must be sure that an FPU wait call is inserted in the microcontroller code immediately after the user-defined function call. If there are other instructions in the instruction buffer, or another instruction is sent before the `EXTWAIT` procedure has completed, it will terminate and return.

**Examples**

```
TIMESET(0) ; clear the internal timer
EXTSET(0) ; clear the external input counter

EXTWAIT ; wait for the next external input
usec = TICKLONG() ; get the elapsed time
```

**See Also**

`EXTLONG`, `EXTSET`

*`uM-FPU64 Instruction Set: EXTWAIT`*
**FCNV**

Converts a floating point value using one of the built-in conversions.

**Syntax**

\[ \text{result} = \text{FCNV}(\text{value}, \text{conversion}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>float</td>
<td>The converted value.</td>
</tr>
<tr>
<td>value</td>
<td>float expression</td>
<td>The value to convert.</td>
</tr>
<tr>
<td>conversion</td>
<td>long constant</td>
<td>The conversion number or conversion symbol. (see list below)</td>
</tr>
</tbody>
</table>

**Notes**

The FCNV function has pre-defined symbols for all conversion numbers as shown in the table below. If the conversion number is out of range, the value is returned with no conversion.

<table>
<thead>
<tr>
<th>Conversion Number</th>
<th>Conversion Symbol</th>
<th>Description</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F_C</td>
<td>Fahrenheit to Celsius</td>
<td>result = value * 1.8 + 32</td>
</tr>
<tr>
<td>1</td>
<td>C_F</td>
<td>Celsius to Fahrenheit</td>
<td>result = (value - 32) / 1.8</td>
</tr>
<tr>
<td>2</td>
<td>IN_MM</td>
<td>inches to millimeters</td>
<td>result = value * 25.4</td>
</tr>
<tr>
<td>3</td>
<td>MM_IN</td>
<td>millimeters to inches</td>
<td>result = value / 25.4</td>
</tr>
<tr>
<td>4</td>
<td>IN_CM</td>
<td>inches to centimeters</td>
<td>result = value * 2.54</td>
</tr>
<tr>
<td>5</td>
<td>CM_IN</td>
<td>centimeters to inches</td>
<td>result = value / 2.54</td>
</tr>
<tr>
<td>6</td>
<td>IN_M</td>
<td>inches to meters</td>
<td>result = value * 0.0254</td>
</tr>
<tr>
<td>7</td>
<td>M_IN</td>
<td>meters to inches</td>
<td>result = value / 0.0254</td>
</tr>
<tr>
<td>8</td>
<td>FT_M</td>
<td>feet to meters</td>
<td>result = value * 0.3048</td>
</tr>
<tr>
<td>9</td>
<td>M_FT</td>
<td>meters to feet</td>
<td>result = value / 0.3048</td>
</tr>
<tr>
<td>10</td>
<td>YD_M</td>
<td>yards to meters</td>
<td>result = value * 0.9144</td>
</tr>
<tr>
<td>11</td>
<td>M_YD</td>
<td>meters to yards</td>
<td>result = value / 0.9144</td>
</tr>
<tr>
<td>12</td>
<td>MILES_KM</td>
<td>miles to kilometers</td>
<td>result = value * 1.609344</td>
</tr>
<tr>
<td>13</td>
<td>KM_MILES</td>
<td>kilometers to miles</td>
<td>result = value / 1.609344</td>
</tr>
<tr>
<td>14</td>
<td>NM_M</td>
<td>nautical miles to meters</td>
<td>result = value * 1852.0</td>
</tr>
<tr>
<td>15</td>
<td>M_NM</td>
<td>meters to nautical miles</td>
<td>result = value / 1852.0</td>
</tr>
<tr>
<td>16</td>
<td>ACRES_M2</td>
<td>acres to meters(^2)</td>
<td>result = value * 4046.856422</td>
</tr>
<tr>
<td>17</td>
<td>M2_ACRES</td>
<td>meters(^2) to acres</td>
<td>result = value / 4046.856422</td>
</tr>
<tr>
<td>18</td>
<td>OZ_G</td>
<td>ounces to grams</td>
<td>result = value * 28.34952313</td>
</tr>
<tr>
<td>19</td>
<td>G_OZ</td>
<td>grams to ounces</td>
<td>result = value / 28.34952313</td>
</tr>
<tr>
<td>20</td>
<td>LB_KG</td>
<td>pounds to kilograms</td>
<td>result = value * 0.45359237</td>
</tr>
<tr>
<td>21</td>
<td>KG_LB</td>
<td>kilograms to pounds</td>
<td>result = value / 0.45359237</td>
</tr>
<tr>
<td>22</td>
<td>USGAL_L</td>
<td>US gallons to liters</td>
<td>result = value * 3.785411784</td>
</tr>
<tr>
<td>23</td>
<td>L_USGAL</td>
<td>liters to US gallons</td>
<td>result = value / 3.785411784</td>
</tr>
<tr>
<td>24</td>
<td>UKGAL_L</td>
<td>UK gallons to liters</td>
<td>result = value * 4.546099295</td>
</tr>
<tr>
<td>25</td>
<td>L_UKGAL</td>
<td>liters to UK gallons</td>
<td>result = value / 4.546099295</td>
</tr>
<tr>
<td>26</td>
<td>USOZ_ML</td>
<td>US fluid ounces to milliliters</td>
<td>result = value * 29.57352956</td>
</tr>
<tr>
<td>27</td>
<td>ML_USOZ</td>
<td>milliliters to US fluid ounces</td>
<td>result = value / 29.57352956</td>
</tr>
</tbody>
</table>
### uM-FPU64 Instruction Set: FCNV

<table>
<thead>
<tr>
<th>Example Description</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert UK fluid ounces to milliliters</td>
<td>( \text{result} = \text{value} \times 28.41312059 )</td>
</tr>
<tr>
<td>Convert milliliters to UK fluid ounces</td>
<td>( \text{result} = \text{value} / 28.41312059 )</td>
</tr>
<tr>
<td>Convert calories to Joules</td>
<td>( \text{result} = \text{value} \times 4.18605 )</td>
</tr>
<tr>
<td>Convert Joules to calories</td>
<td>( \text{result} = \text{value} / 4.18605 )</td>
</tr>
<tr>
<td>Convert horsepower to watts</td>
<td>( \text{result} = \text{value} \times 745.7 )</td>
</tr>
<tr>
<td>Convert watts to horsepower</td>
<td>( \text{result} = \text{value} / 745.7 )</td>
</tr>
<tr>
<td>Convert atmospheres to kilopascals</td>
<td>( \text{result} = \text{value} \times 101.325 )</td>
</tr>
<tr>
<td>Convert kilopascals to atmospheres</td>
<td>( \text{result} = \text{value} / 101.325 )</td>
</tr>
<tr>
<td>Convert mmHg to kilopascals</td>
<td>( \text{result} = \text{value} \times 133.3223684 )</td>
</tr>
<tr>
<td>Convert kilopascals to mmHg</td>
<td>( \text{result} = \text{value} / 133.3223684 )</td>
</tr>
<tr>
<td>Convert degrees to radians</td>
<td>( \text{result} = \text{value} \times \pi / 180 )</td>
</tr>
<tr>
<td>Convert radians to degrees</td>
<td>( \text{result} = \text{value} \times 180 / \pi )</td>
</tr>
</tbody>
</table>

#### Examples

- \( \text{distance} = \text{FCNV}(200, \text{FT}_M) \); returns 60.96 (meters)
- \( \text{tempF} = \text{FCNV}(100, \text{C}_F) \); returns 212.0 (degree fahrenheit)
- \( \text{tempF} = \text{FCNV}(100, 1) \); returns 212.0 (degree fahrenheit)

#### See Also

*uM-FPU64 Instruction Set: FCNV*
FFT

Perform a Fast Fourier Transform.

Syntax

\[ \text{FFT} \left( \text{type} \right) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>long constant</td>
<td>The type of FFT operation:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FIRST_STAGE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NEXT_STAGE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NEXT_LEVEL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NEXT_BLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modifiers:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+REVERSE bit reverse sort pre-processing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+PRE pre-processing for inverse FFT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+POST post-processing for inverse FFT</td>
</tr>
</tbody>
</table>

Notes

The data for the FFT instruction is stored in matrix A as a Nx2 matrix, where N must be a power of two. The data points are specified as complex numbers, with the real part stored in the first column and the imaginary part stored in the second column. If all data points can be stored in the matrix (maximum of 64 points if all 128 registers are used), the Fast Fourier Transform can be calculated with a single instruction. If more data points are required than will fit in the matrix, the calculation must be done in blocks. The algorithm iteratively writes the next block of data, executes the FFT instruction for the appropriate stage of the FFT calculation, and reads the data back to the microcontroller. This proceeds in stages until all data points have been processed.

See Application Note 35 - Fast Fourier Transforms using the FFT Instruction for more details.

Examples

\[ \text{FFT}(\text{FIRST_STAGE}+\text{REVERSE}) ; \text{perform FFT in single instruction} \]

See Also

*um-FPU64 Instruction Set: FFT*
FLOOKUP

Returns a floating point value from a lookup table.

*Note:* Must be used inside a user-defined procedure or function.

**Syntax**

```plaintext
result = FLOOKUP(value, item0, item1, ...)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>float</td>
<td>The returned value.</td>
</tr>
<tr>
<td>value</td>
<td>long expression</td>
<td>The lookup index for the lookup table.</td>
</tr>
<tr>
<td>item0,</td>
<td>float constant</td>
<td>The list of floating point constants for the lookup table. A maximum of 256 values are allowed.</td>
</tr>
<tr>
<td>item1, ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

The lookup index is used to return the corresponding item from the lookup table. The items are indexed sequentially starting at zero. A value of zero is returned if the index is less than zero or greater than the number of entries in the table.

**Examples**

```plaintext
result = FLOOKUP(n, 0, 1.0, 10.0, 100, 1000) ; if n = 2, then 10.0 is returned
```

**See Also**

`FTABLE, LLOOKUP, LTABLE`

*uM-FPU64 Instruction Set: TABLE*
FOR...NEXT

Executes a group of statements a specified number of times.

Note: Must be used inside a user-defined procedure or function.

Syntax

```
FOR register = startExpression TO | DOWNTO endExpression [STEP stepExpression]
[statements]
[CONTINUE]
(EXIT)
NEXT
```

Notes

Before the FOR loop begins, the register is set to the value of `startExpression`. At the start of each FOR loop, the `register` value is compared to the `endExpression` value. If `TO` is used, and the `register` value is greater than the `endExpression` value, the FOR loop is terminated. If `DOWNTO` is used, and the `register` value is less than the `endExpression` value, the FOR loop is terminated. If the FOR loop does not terminate, the statements in the FOR loop are executed. When the NEXT statement is encountered, the value of `stepExpression` is added to the `register` if `TO` is used, or subtracted from the `register` if `DOWNTO` is used, and execution returns to the start of the FOR loop. If the `STEP` clause is not included, `stepExpression` is 1. The `stepExpression` must be a positive value for the loop to terminate. The `CONTINUE` statement is used to skip ahead to the NEXT statement. The `EXIT` statement is used to immediately terminate the FOR loop. The `RETURN` statement is used to exit the user-defined function.

Examples

```
n equ L10
x equ F11

FOR x = 1 to 10 STEP 0.5 ; x = 1.0, 1.5, 2.0, ..., 10.0
    ; statements executed each loop iteration
    if n > 1500 then EXIT
NEXT
```
n equ L10
x equ F11

FOR n = 10 DOWNTO 1 ; n = 10, 9, 8, ..., 1
    ; statements executed each loop iteration
    if x > 1500 then CONTINUE
    ; statements only executed if x <= 1500
NEXT

See Also
CONTINUE, DO...WHILE...UNTIL...LOOP, EXIT, IF...THEN, IF...THEN...ELSE,
RETURN, SELECT...CASE
FTABLE

Returns the index of the first item in the list that satisfies the condition code.

Note: Must be used inside a user-defined procedure or function.

Syntax

\[
\text{result} = \text{FTABLE}(\text{value}, \text{cc}, \text{item0}, \text{item1}, \ldots)
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The index of the first item in the lookup table that satisfies the condition.</td>
</tr>
<tr>
<td>value</td>
<td>float expression</td>
<td>The floating point value to compare with the table items.</td>
</tr>
<tr>
<td>cc</td>
<td>condition code</td>
<td>Condition code. Z, NZ, EQ, NE, LT, GE, LE, GT</td>
</tr>
<tr>
<td>item0,</td>
<td>item1, ...</td>
<td>float constant. A list of floating point constants for the lookup table. A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>maximum of 256 values are allowed.</td>
</tr>
</tbody>
</table>

Notes

The specified value is compared to each value in the table, and the index value is returned for the first item that satisfies the condition code. The index value starts at zero.

Examples

If the condition code is GE, then the items in the list are compared as follows:

\[
\begin{align*}
\text{value} & \geq \text{item0} \\
\text{value} & \geq \text{item1} \\
\text{value} & \geq \text{item2} \\
\ldots
\end{align*}
\]

\[
\text{index} = \text{FLOOKUP(\text{value}, \text{GE}, 1.0, 5.5, 10.0, 100.0)}
\]

; if value = 1, index = 0

; if value = 17.5, index = 2

See Also

FLOOKUP, LLOOKUP, LTABLE

uM-FPU64 Instruction Set: FTABLE
FTOA

Convert floating point value to string.

Syntax

\[ \text{FTOA}(\text{value}, \text{format}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>float expression</td>
<td>The floating point value to convert.</td>
</tr>
<tr>
<td>format</td>
<td>long constant</td>
<td>The format specifier.</td>
</tr>
</tbody>
</table>

Notes

The floating point value is converted to a string and stored at the string selection point. The selection point is updated to point immediately after the inserted string, so multiple insertions can be appended.

If the format byte is zero, as many digits as necessary will be used to represent the number with up to eight significant digits. Very large or very small numbers are represented in exponential notation. The length of the displayed value is variable and can be from 3 to 12 characters in length. The special cases of NaN (Not a Number), +infinity, -infinity, and -0.0 are handled. Examples of the ASCII strings produced are as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>NaN</td>
<td>0.0</td>
</tr>
<tr>
<td>10e20</td>
<td>Infinity</td>
<td>-0.0</td>
</tr>
<tr>
<td>3.1415927</td>
<td>-Infinity</td>
<td>1.0</td>
</tr>
<tr>
<td>-52.333334</td>
<td>-3.5e-5</td>
<td>0.01</td>
</tr>
</tbody>
</table>

If the format byte is non-zero, it is interpreted as a decimal number. The tens digit specifies the maximum length of the converted string, and the ones digit specifies the number of decimal points. The maximum number of digits for the formatted conversion is 9, and the maximum number of decimal points is 6. If the floating point value is too large for the format specified, asterisks will be stored. If the number of decimal points is zero, no decimal point will be displayed. Examples of the display format are as follows: (note: leading spaces are shown where applicable)

<table>
<thead>
<tr>
<th>Value in register A</th>
<th>Format byte</th>
<th>Display format</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.567</td>
<td>61 (6.1)</td>
<td>123.6</td>
</tr>
<tr>
<td>123.567</td>
<td>62 (6.2)</td>
<td>123.57</td>
</tr>
<tr>
<td>123.567</td>
<td>42 (4.2)</td>
<td><strong>.</strong></td>
</tr>
<tr>
<td>0.9999</td>
<td>20 (2.0)</td>
<td>1</td>
</tr>
<tr>
<td>0.9999</td>
<td>31 (3.1)</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Examples
In the following example the [] characters are used to show the string selection point.

```
x    equ F10
STRSET(""") ; string buffer = []
FTOA(pi, 0) ; string buffer = 3.1415927[
STRINS("","") ; string buffer = 3.1415927,[
x = 2/3
FTOA(x, 63) ; string buffer = 3.1415927, 0.667[
```

See Also
LTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSEL, STRSET

*uM-FPU64 Instruction Set: STRINC, STRDEC*
**IF...THEN**

Conditionally executes a statement.

*Note:* Must be used inside a user-defined procedure or function.

**Syntax**

```
IF condition THEN CONTINUE
IF condition THEN EXIT
IF condition THEN RETURN
IF condition THEN equalsStatement
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>A conditional expression.</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>The statement is executed if <code>condition</code> is true.</td>
</tr>
<tr>
<td>EXIT</td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td>equalsStatement</td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

If the condition is true, the statement is executed.

**Examples**

```
if sin(angle) < 0.3 then n = 0

if n then return ; if n is not zero, then return

for n = 1 to 10
  ;...
  if m < 0 then exit ; if m is less than zero, then exit from for loop
next
```
**IF...THEN...ELSE**

Conditionally executes a statement or group of statements.

*Note:* Must be used inside a user-defined procedure or function.

**Syntax**

```plaintext
IF condition THEN
    statements
[ELSEIF condition THEN
    statements]...
[ELSE
    statements]
ENDIF
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>A conditional expression.</td>
</tr>
<tr>
<td>statements</td>
<td>One or more statements that execute if condition is true.</td>
</tr>
</tbody>
</table>

**Notes**

If the IF condition is true, then the statements following the THEN clause are executed. If the IF condition is false, then any ELSEIF clauses that are included are tested in sequence. If an ELSEIF condition is true, the statements associated with that ELSEIF clause are executed. If no IF or ELSEIF conditions are true, and an ELSE clause is included, the statements in the ELSE clause are executed.

**Examples**

```plaintext
if n > 0 then
    m = 1
elseif n < 0 then
    m = -1
else
    m = 0
next
```

**See Also**

*Conditional Expressions, DO...WHILE...UNTIL...LOOP, FOR...NEXT, IF...THEN, SELECT...CASE*
**Line Continuation**

The underscore character (_ _) is used as a line continuation character. The underscore must be the last character on the line, other than whitespace characters or comments. The underscore character must not be placed in the middle of a number, symbol name or string literal.

**Examples**

```
result = FLOOKUP(n, 0.0, 1000.0, 2000.0, _, 3000.0, 4000.0) ; first line
```

```
 ; line continuation
```
**LLOOKUP**

Returns a long integer value from a lookup table.

*Note:* Must be used inside a user-defined procedure or function.

**Syntax**

```
result = LLOOKUP(value, item0, item1, ...)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The returned value.</td>
</tr>
<tr>
<td>value</td>
<td>long expression</td>
<td>The lookup index.</td>
</tr>
<tr>
<td>item0, item1, ...</td>
<td>long constant</td>
<td>The list of long integer constants in the table. A maximum of 256 values are allowed.</td>
</tr>
</tbody>
</table>

**Notes**

The lookup index is used to return the corresponding item from the lookup table. The items are indexed sequentially starting at zero. A value of zero is returned if the index is less than zero or greater than the number of entries in the table.

**Examples**

```
result = LLOOKUP(n, 0, 1, 10, 100, 1000); if n = 2, then result = 10.0
```

**See Also**

FLOOKUP, FTABLE, LTABLE

*uM-FPU64 Instruction Set: TABLE*
LOADMA
LOADMB
LOADMC

Returns the value of an element in the specified matrix. LOADMA accesses matrix A, LOADMB accesses matrix B, and LOADMC accesses matrix C.

Syntax

result = LOADMA(row, column)
result = LOADMB(row, column)
result = LOADMC(row, column)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>float</td>
<td>The value of the selected matrix element.</td>
</tr>
<tr>
<td>row</td>
<td>long constant</td>
<td>register</td>
</tr>
<tr>
<td>column</td>
<td>long constant</td>
<td>register</td>
</tr>
</tbody>
</table>

Notes

The row and column numbers are used to select the element of the matrix. The row and column numbers start from zero. If the row or column values are out of range, NaN is returned.

Examples

```plaintext
value = LOADMA(1,2) ; get the value at row 1, column 2 of matrix A
```

See Also

MOP, SAVEMA, SAVEMB, SAVEMC, SELECTMA, SELECTMB, SELECTMC

uM-FPU64 Instruction Set: LOADMA, LOADMB, LOADMC
**LTABLE**

Returns the index of the first table entry that satisfies the condition code. The specified value is compared to each value in the list of items, and the index value is returned. The index value starts at zero.

*Note:* Must be used inside a user-defined procedure or function.

**Syntax**

```
result = LTABLE(value, cc, item0, item1, ...)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The index of the first table entry that satisfies the condition.</td>
</tr>
<tr>
<td>value</td>
<td>long expression</td>
<td>The long integer value to compare with the table items.</td>
</tr>
<tr>
<td>cc</td>
<td>condition code</td>
<td>Condition code.</td>
</tr>
<tr>
<td>item0, item1, ...</td>
<td>long constant</td>
<td>The list of long integer constants for the lookup table.</td>
</tr>
</tbody>
</table>

**Notes**

The specified value is compared to each value in the table, and the index value is returned for the first item that satisfies the condition code. The index value starts at zero.

**Examples**

If the condition code is LT, then the items in the list are compared as follows:

```
value < item0
value < item1
value < item2
...
```

```
index = LLOOKUP(value, LT, 1, 50, 1000, 10000); if value = 1, index = 1
        ; if value = 500, index = 2
```

**See Also**

FLOOKUP, FTABLE, LLOOKUP

*uM-FPU64 Instruction Set: LTABLE*
LTOA

Convert long integer value to string.

Syntax

\[
\text{LTOA}(\text{value}, \text{format})
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>long expression</td>
<td>The long integer value to convert.</td>
</tr>
<tr>
<td>format</td>
<td>long constant</td>
<td>The format specifier.</td>
</tr>
</tbody>
</table>

Notes

The long integer value is converted to a string and stored at the string selection point. The selection point is updated to point immediately after the inserted string, so multiple insertions can be appended.

If \(\text{format}\) is zero, the length of the converted string is variable and can range from 1 to 11 characters in length. Examples of the converted string are as follows:

1
500000
\(-3598390\)

If \(\text{format}\) is non-zero, a value between 0 and 15 specifies the length of the converted string. The converted string is right justified. If \(\text{format}\) is positive, its absolute value specifies the length of the converted string, and leading zeros are used. If 100 is added to the \(\text{format}\) value the value is converted as an unsigned long integer, otherwise it is converted as an signed long integer. If the converted string is longer than the specified length, asterisks are stored. If the length is specified as zero, the string will be as long as necessary to represent the number. Examples of the converted string are as follows: (note: leading spaces are shown where applicable)

<table>
<thead>
<tr>
<th>Value in register A</th>
<th>Format byte</th>
<th>Description</th>
<th>Display format</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>10</td>
<td>signed, length = 10</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>110</td>
<td>unsigned, length = 10</td>
<td>4294967295</td>
</tr>
<tr>
<td>-1</td>
<td>4</td>
<td>signed, length = 4</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>104</td>
<td>unsigned, length = 4</td>
<td>****</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>signed, length = 4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>unformatted</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>6</td>
<td>signed, length = 6</td>
<td>1000</td>
</tr>
<tr>
<td>1000</td>
<td>-6</td>
<td>signed, length = 6, zero fill</td>
<td>001000</td>
</tr>
</tbody>
</table>
**Examples**

```assembly
year   equ  L10
month  equ  L11
day    equ  L11

year = 2010
month = 7
day = 20
STRSET("Date stamp: "); string buffer = Date stamp: []
LTOA(year, 0); string buffer = Date stamp: 2010[]
STRINS("-"); string buffer = Date stamp: 2010-[]
LTOA(month, 0); string buffer = Date stamp: 2010-7[]
STRINS("-"); string buffer = Date stamp: 2010-7-[]
LTOA(day, 0); string buffer = Date stamp: 2010-7-20[]
```

**See Also**

FTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSEL, STRSET

*uM-FPU64 Instruction Set: STRINC, STRDEC*
**Math Functions**

All of the math functions supported in the previous version of the IDE are still supported.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Arguments</th>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = SQRT(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Square root of arg1.</td>
</tr>
<tr>
<td>result = LOG(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Logarithm (base e) of arg1.</td>
</tr>
<tr>
<td>result = LOG10(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Logarithm (base 10) of arg1.</td>
</tr>
<tr>
<td>result = EXP(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>e to the power of arg1.</td>
</tr>
<tr>
<td>result = EXP10(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>10 to the power of arg1.</td>
</tr>
<tr>
<td>result = SIN(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Sine of the angle arg1 (in radians).</td>
</tr>
<tr>
<td>result = COS(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Cosine of the angle arg1 (in radians).</td>
</tr>
<tr>
<td>result = TAN(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Tangent of the angle arg1 (in radians).</td>
</tr>
<tr>
<td>result = ASIN(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Inverse sine of the value arg1.</td>
</tr>
<tr>
<td>result = ACOS(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Inverse cosine of the value arg1.</td>
</tr>
<tr>
<td>result = ATAN(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Inverse tangent of the value arg1.</td>
</tr>
<tr>
<td>result = ATAN2(arg1, arg2)</td>
<td>Float</td>
<td>Float</td>
<td>Inverse tangent of the value arg1 divided by arg2.</td>
</tr>
<tr>
<td>result = DEGREES(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Converts angle arg1 from radians to degrees.</td>
</tr>
<tr>
<td>result = RADIANS(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Converts angle arg1 from degrees to radians.</td>
</tr>
<tr>
<td>result = FLOOR(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Floor of arg1.</td>
</tr>
<tr>
<td>result = CEIL(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Ceiling of arg1.</td>
</tr>
<tr>
<td>result = ROUND(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>arg1 rounded to the nearest integer.</td>
</tr>
<tr>
<td>result = POWER(arg1, arg2)</td>
<td>Float</td>
<td>Float</td>
<td>arg1 raised to the power of arg2.</td>
</tr>
<tr>
<td>result = ROOT(arg1, arg2)</td>
<td>Float</td>
<td>Float</td>
<td>arg2 root of arg1.</td>
</tr>
<tr>
<td>result = FRAC(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Fractional part of arg1.</td>
</tr>
<tr>
<td>result = FLOAT(arg1)</td>
<td>Long</td>
<td>Float</td>
<td>Converts arg1 from long to float.</td>
</tr>
<tr>
<td>result = FIX(arg1)</td>
<td>Float</td>
<td>Long</td>
<td>Converts arg1 from float to long.</td>
</tr>
<tr>
<td>result = FIXR(arg1)</td>
<td>Float</td>
<td>Long</td>
<td>Rounds arg1 then converts from float to long.</td>
</tr>
<tr>
<td>result = ABS(arg1)</td>
<td>Float</td>
<td>Float</td>
<td>Absolute value of arg1.</td>
</tr>
<tr>
<td>result = ABS(arg1)</td>
<td>Long</td>
<td>Long</td>
<td>Absolute value of arg1.</td>
</tr>
<tr>
<td>result = MOD(arg1, arg2)</td>
<td>Float</td>
<td>Float</td>
<td>Remainder of arg1 divided by arg2.</td>
</tr>
<tr>
<td>result = MOD(arg1, arg2)</td>
<td>Long</td>
<td>Long</td>
<td>Remainder of arg1 divided by arg2.</td>
</tr>
<tr>
<td>result = MIN(arg1, arg2)</td>
<td>Float</td>
<td>Float</td>
<td>Minimum of arg1 and arg2.</td>
</tr>
<tr>
<td>result = MIN(arg1, arg2)</td>
<td>Long</td>
<td>Long</td>
<td>Minimum of arg1 and arg2.</td>
</tr>
<tr>
<td>result = MAX(arg1, arg2)</td>
<td>Float</td>
<td>Float</td>
<td>Maximum of arg1 and arg2.</td>
</tr>
<tr>
<td>result = MAX(arg1, arg2)</td>
<td>Long</td>
<td>Long</td>
<td>Maximum of arg1 and arg2.</td>
</tr>
</tbody>
</table>

**Examples**

```plaintext
theta = sin(angle)
result = cos(PI/2 + sin(theta))
```

**See Also**

*uM-FPU64 Instruction Set:* 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).
MOP

Performs matrix operations. The matrix operations are summarized below.

\[
\begin{align*}
\text{MOP} & (\text{SCALAR\_SET}, \text{value}) \\
\text{MOP} & (\text{SCALAR\_ADD}, \text{value}) \\
\text{MOP} & (\text{SCALAR\_SUB}, \text{value}) \\
\text{MOP} & (\text{SCALAR\_SUBR}, \text{value}) \\
\text{MOP} & (\text{SCALAR\_MUL}, \text{value}) \\
\text{MOP} & (\text{SCALAR\_DIV}, \text{value}) \\
\text{MOP} & (\text{SCALAR\_DIVR}, \text{value}) \\
\text{MOP} & (\text{SCALAR\_POW}, \text{value}) \\
\text{MOP} & (\text{EWISE\_SET}) \\
\text{MOP} & (\text{EWISE\_ADD}) \\
\text{MOP} & (\text{EWISE\_SUB}) \\
\text{MOP} & (\text{EWISE\_SUBR}) \\
\text{MOP} & (\text{EWISE\_MUL}) \\
\text{MOP} & (\text{EWISE\_DIV}) \\
\text{MOP} & (\text{EWISE\_DIVR}) \\
\text{MOP} & (\text{EWISE\_POW}) \\
\text{MOP} & (\text{MULTIPLY}) \\
\text{MOP} & (\text{IDENTITY}) \\
\text{MOP} & (\text{DIAGONAL}, \text{value}) \\
\text{MOP} & (\text{TRANSPOSE}) \\
\text{return} & = \text{MOP}(\text{COUNT}) \\
\text{return} & = \text{MOP}(\text{SUM}) \\
\text{return} & = \text{MOP}(\text{AVE}) \\
\text{return} & = \text{MOP}(\text{MIN}) \\
\text{return} & = \text{MOP}(\text{MAX}) \\
\text{MOP} & (\text{COPY\_AB}) \\
\text{MOP} & (\text{COPY\_AC}) \\
\text{MOP} & (\text{COPY\_BA}) \\
\text{MOP} & (\text{COPY\_BC}) \\
\text{MOP} & (\text{COPY\_CA}) \\
\text{MOP} & (\text{COPY\_CB}) \\
\text{return} & = \text{MOP}(\text{DETERMINANT}) \\
\text{MOP} & (\text{INVERSE}) \\
\text{MOP} & (\text{LOAD\_RA}, idx1, idx2, ...) \\
\text{MOP} & (\text{LOAD\_RB}, idx1, idx2, ...) \\
\text{MOP} & (\text{LOAD\_RC}, idx1, idx2, ...) \\
\text{MOP} & (\text{LOAD\_BA}, idx1, idx2, ...) \\
\text{MOP} & (\text{LOAD\_CA}, idx1, idx2, ...) \\
\text{MOP} & (\text{SAVE\_AR}, idx1, idx2, ...) \\
\text{MOP} & (\text{SAVE\_AB}, idx1, idx2, ...) \\
\text{MOP} & (\text{SAVE\_AC}, idx1, idx2, ...) \\
\text{MOP} & (\text{LU\_DECOMP}) \\
\text{MOP} & (\text{LU\_INVERSE}) \\
\text{MOP} & (\text{LU\_DETERM}) \\
\text{MOP} & (\text{LU\_SOLVE}) \\
\text{MOP} & (\text{CH\_DECOMP}) \\
\text{MOP} & (\text{CH\_INVERSE}) \\
\text{MOP} & (\text{CH\_DETERM}) \\
\text{MOP} & (\text{CH\_SOLVE})
\end{align*}
\]

A detailed description of each MOP operation is shown below.
Syntax

MOP(SCALAR_SET, value)
MOP(SCALAR_ADD, value)
MOP(SCALAR_SUB, value)
MOP(SCALAR_SUBR, value)
MOP(SCALAR_MUL, value)
MOP(SCALAR_DIV, value)
MOP(SCALAR_DIVR, value)
MOP(SCALAR_POW, value)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>float expression</td>
<td>The scalar value used for the matrix operation.</td>
</tr>
</tbody>
</table>

Notes

The scalar operations apply the specified value to each element of matrix A as follows:

SCALAR_SET  Set each element of matrix A to the specified value.
MA[row, column] = value

SCALAR_ADD  Add the specified value to each element of matrix A.
MA[row, column] = MA[row, column] + value

SCALAR_SUB  Subtract the specified value from each element of matrix A.
MA[row, column] = MA[row, column] - value

SCALAR_SUBR Subtract the value of each element of matrix A from the specified value.
MA[row, column] = value - MA[row, column]

SCALAR_MUL  Multiply each element of matrix A by the specified value.
MA[row, column] = MA[row, column] * value

SCALAR_DIV  Divide each element of matrix A by the specified value.
MA[row, column] = MA[row, column] / value

SCALAR_DIVR Divide the specified value by each element in matrix A.
MA[row, column] = value / MA[row, column]

SCALAR_POW  Each element of matrix A is raised to the power of the specified value.
MA[row, column] = MA[row, column] ** value

Examples

MOP(SCALAR_SET, 1.0) ; sets all elements of matrix A to 1.0
MOP(SCALAR_MUL, scale) ; multiplies all elements of matrix A by the value of scale
Syntax

MOP(ewise_set)
MOP(ewise_add)
MOP(ewise_sub)
MOP(ewise_subr)
MOP(ewise_mul)
MOP(ewise_div)
MOP(ewise_divr)
MOP(ewise_pow)

Notes
The element-wise operations perform their operations using corresponding elements from matrix A and matrix B and store the result in matrix A. Element-wise operations are only performed if both matrices must have the same number of rows and columns. The operations are as follows:

ewise_set Set each element of matrix A to the value of the element in matrix B.
MA[row, column] = MB[row, column]
ewise_add Add the value of each element of matrix B to the element of matrix A.
MA[row, column] = MA[row, column] + MB[row, column]
ewise_sub Subtract the value of each element of matrix B from the element of matrix A.
MA[row, column] = MA[row, column] - MB[row, column]
ewise_subr Subtract the value of each element of matrix A from the element of matrix B.
MA[row, column] = MB[row, column] - MA[row, column]
ewise_mul Multiply each element of matrix A by the element of matrix B.
MA[row, column] = MA[row, column] * MB[row, column]
ewise_div Divide each element of matrix A by the element of matrix B.
MA[row, column] = MA[row, column] / MB[row, column]
ewise_divr Divide each element of matrix B by the element of matrix A.
ewise_pow Each element of matrix A is raised to the power of the element of matrix B.
MA[row, column] = MA[row, column] ** MB[row, column]

Examples

MOP(ewise_div) ; each elements of matrix A is divided by the element in matrix B

Syntax

MOP(multiply)

Notes
Performs a matrix multiplication. Matrix B is multiplied by matrix C and the result is stored in matrix A. The matrix multiply is only performed if the number of columns in matrix B is the same as the number of rows in matrix C. The size of matrix MA will be updated to reflect the rows and columns of the resulting matrix.
Examples

\[ \text{MOP(MULTIPLY)} \] ; multiplies matrix A by matrix B

Syntax

\[ \text{MOP(IDENTITY)} \]

Notes
Sets matrix A to the identity matrix. The identity matrix has the value 1.0 stored on the diagonal and all others elements are set to zero.

Examples

\[ \text{MOP(IDENTITY)} \] ; sets matrix A to the identity matrix

Syntax

\[ \text{MOP(DIAGONAL, value)} \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>float expression</td>
<td>The value to store on the diagonal.</td>
</tr>
</tbody>
</table>

Notes
Sets matrix A to a diagonal matrix. The specified value is stored on the diagonal and all others elements are set to zero.

Examples

\[ \text{MOP(DIAGONAL, 100.0)} \] ; set matrix A to a diagonal matrix with 100.0 stored on the diagonal

Syntax

\[ \text{MOP(TRANSPOSE)} \]

Notes
Sets matrix A to the transpose of matrix B.
Examples

```
MOP (TRANPOSE) ; sets matrix A to the transpose of matrix B
```

Syntax

```
return = MOP (COUNT)
return = MOP (SUM)
return = MOP (AVE)
return = MOP (MIN)
return = MOP (MAX)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>float</td>
<td>COUNT - number of elements</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>SUM - sum of all elements</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>AVE - average of all elements</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>MIN - minimum value of all elements</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>MAX - maximum value of all elements</td>
</tr>
</tbody>
</table>

Notes

Performs statistical calculations. The value returned is the count, sum, average, minimum, or maximum of all elements in matrix A.

Examples

```
SELECTMA (array, 3, 3) ; set matrix A as 3x3 array
MOP (SCALAR_SET, 0) ; set all values to zero
SAVEMA (1, 1, 10.0) ; store 10.0 at array(1,1)
n=MOP (COUNT) ; returns 9 (the number of elements)
maxValue=MOP (MAX) ; returns 10.0 (the maximum value in array)
```

Syntax

```
MOP (COPY_AB)
MOP (COPY_AC)
MOP (COPY_BA)
MOP (COPY_BC)
MOP (COPY_CA)
MOP (COPY_CB)
```

Notes

Copies one matrix to another.

Examples

```
MOP (COPY_AB) ; copies matrix A to matrix B
```
Syntax

\[ \text{return} = \text{MOP}(\text{DETERMINANT}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{return}</td>
<td>float</td>
<td>The determinant of matrix A.</td>
</tr>
</tbody>
</table>

Notes
Calculates the determinant of matrix A. Matrix A must be a 2x2 or 3x3 matrix.

Examples

\[ \text{value} = \text{MOP}(\text{DETERMINANT}) \quad ; \text{return the determinant of matrix A} \]

Syntax

\[ \text{MOP}(\text{INVERSE}) \]

Notes
The inverse of matrix B is stores as matrix A. Matrix B must be a 2x2 or 3x3 matrix.

Examples

\[ \text{MOP}(\text{INVERSE}) \quad ; \text{sets matrix A to the inverse of matrix B} \]

Syntax

\[ \text{MOP}(\text{LOAD\_RA}, \text{idx1}, \text{idx2}, ...) \]
\[ \text{MOP}(\text{LOAD\_RB}, \text{idx1}, \text{idx2}, ...) \]
\[ \text{MOP}(\text{LOAD\_RC}, \text{idx1}, \text{idx2}, ...) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{idx1, idx2, ...}</td>
<td>byte constants</td>
<td>Index values.</td>
</tr>
</tbody>
</table>

Notes
The indexed load register to matrix operations can be used to quickly load a matrix by copying register values to a matrix. Each index value is a signed 8-bit integer specifying one of the registers from 0 to 127. If the index is positive, the value of the indexed register is copied to the matrix. If the index is negative, the absolute value is used as an index, and the negative value of the indexed register is copied to the matrix. Register 0 is cleared to zero before the register values are copied, so index 0 will always store a zero value in the matrix. The values are stored sequentially, beginning with the first register in the destination matrix.

Examples
Suppose you wanted to create a 2-dimensional rotation matrix as follows:
Assuming register 1 contains the value \( \sin A \), and register 2 contains the value \( \cos A \), the following instructions create the matrix.

\[
\begin{array}{cc}
\cos A & -\sin A \\
\sin A & \cos A
\end{array}
\]

```
SELECTMA(array, 2, 2) ; selects matrix A as a 2x2 matrix at the register called array
MOP(LOAD_RA, 2, -1, 1, 2) ; sets matrix A to the rotation matrix shown above
```

**Syntax**

```
MOP(LOAD_BA, idx1, idx2, ...)
MOP(LOAD_CA, idx1, idx2, ...)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx1,</td>
<td>byte constants</td>
<td>Index values.</td>
</tr>
<tr>
<td>idx2, ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes**
The indexed load matrix to matrix operations can be used to quickly copy values from one matrix to another. Each index value is a signed 8-bit integer specifying the offset of the desired matrix element from the start of the matrix. If the index is positive, the matrix element is copied to matrix A. If the index is negative, the absolute value is used as an index, and the negative value of the matrix element is copied to the destination matrix. Register 0 is cleared to zero before the register values are copied, so index 0 will always store a zero value in matrix A. The values are stored sequentially, beginning with the first register in matrix A.

**Examples**
Suppose matrix B is a 3x3 array and you want to create a 2x2 array from the upper left corner as follows:

\[
\begin{array}{ccc}
a & b & c \\
d & e & f \\
g & h & i
\end{array}
\]

```
SELECTMA(oldArray, 3, 3) ; selects matrix A as a 3x3 matrix at the register called oldArray
SELECTMB(newArray, 2, 2) ; selects matrix B as a 2x2 matrix at the register called newArray
MOP(LOAD_BA, 0, 1, 3, 4) ; copies the subset shown above from matrix A to matrix B
```

**Syntax**
MOP(SAVE_AR, idx1, idx2, ...)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx1, idx2, ...</td>
<td>byte constants</td>
<td>Index values.</td>
</tr>
</tbody>
</table>

Notes
The indexed save matrix to register operation can be used to quickly extract values from a matrix. Each index value is a signed 8-bit integer specifying one of the registers from 0 to 127. The values are stored sequentially, beginning with the first element in matrix A. If the index is positive, the matrix value is copied to the indexed register. If the index is negative, the matrix value is not copied.

Examples
Suppose matrix A is a 3x3 matrix containing the following values:

\[
\begin{array}{ccc}
a & b & c \\
d & e & f \\
g & h & i \\
\end{array}
\]

MOP(SAVE_AR, 10, -1, -1, -1, 11, -1, -1, -1, 12) ; saves element a to register 10
; saves element e to register 11
; saves element i to register 12

Syntax
MOP(SAVE_AB, idx1, idx2, ...)
MOP(SAVE_AC, idx1, idx2, ...)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx1, idx2, ...</td>
<td>byte constants</td>
<td>Index values.</td>
</tr>
</tbody>
</table>

Notes
The indexed save matrix to matrix operations can be used to quickly extract values from a matrix. Each index value is a signed 8-bit integer specifying the offset of the desired matrix element from the start of matrix A. The values are stored sequentially in the destination matrix, beginning with the first element in matrix A. If the index is positive, the matrix value is copied to the destination matrix. If the index is negative, the matrix value is not copied.

Syntax
MOP(LU_DECOMP)
MOP(LU_INVERSE)
The LU and Cholesky decomposition operations can be used to calculate a matrix inverse, matrix determinant, and to solve sets of linear equations for $n \times n$ matrices of any size. The maximum size of matrix will be limited by the available registers or RAM for storing the matrices. An augmented matrix is created by the MOP (LU_DECOMP) and MOP (CH_DECOMP) procedures.

See Also
LOADMA, LOADMB, LOADMC, SAVEMA, SAVEMB, SAVEMC, SELECTMA, SELECTMB, SELECTMC

uM-FPU64 Instruction Set: MOP
POLY

Calculates the $n^{th}$ order polynomial of the floating point value.

Note: Must be used inside a user-defined procedure or function.

Syntax

$$result = \text{POLY}(value, \text{coeff1}, \text{coeff2}, ...)$$

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>float</td>
<td>The result of the $n^{th}$ order polynomial equation.</td>
</tr>
<tr>
<td>value</td>
<td>float expression</td>
<td>The value of $x$ in the polynomial equation.</td>
</tr>
<tr>
<td>coeff1, coeff2, ...</td>
<td>long constant</td>
<td>The coefficient values the polynomial equation. Specified in order from $A_n$ to $A_0$.</td>
</tr>
</tbody>
</table>

Notes

The POLY function can only be used inside an FPU function. The general form of the polynomial is:

$$A_0 + A_1 x^1 + A_2 x^2 + \ldots A_n x^n$$

The coefficients are specified from the highest order $A_n$ to the lowest order $A_0$. If one of the terms is not used in the polynomial, a zero value must be stored in its place.

Examples

```
value = POLY(x, 3.0, 5.0) ; value = 3x + 5
value = POLY(x, 1, 0, 0, 1) ; value = $x^3 + 1$
```

The formula used to compensate for the non-linearity of the SHT1x/SHT7x humidity sensor is a second order polynomial. The formula is as follows:

$$RH_{linear} = -4.0 + 0.0405 \cdot SO_{RH} + (-2.8 \cdot 10^{-6} \cdot SO_{RH}^2)$$

The following example makes this calculation.

```
RH_{linear} = POLY(SO_{Rh}, -2.8E-6, 0.0405, -4)
```

See Also

uM-FPU64 Instruction Set: POLY
READVAR

Returns the value of the selected FPU internal register.

Syntax

\[ \text{result} = \text{READVAR}(\text{number}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The FPU internal register value.</td>
</tr>
<tr>
<td>number</td>
<td>byte constant</td>
<td>The internal variable number. (see list below)</td>
</tr>
</tbody>
</table>

Notes

Internal Variable Number | Description
-------------------------|--------------------------------------------------|
0                        | A register.                                      |
1                        | X register.                                      |
2                        | Matrix A register.                               |
3                        | Matrix A rows.                                   |
4                        | Matrix A columns.                                |
5                        | Matrix B register.                               |
6                        | Matrix B rows.                                   |
7                        | Matrix B columns.                                |
8                        | Matrix C register.                               |
9                        | Matrix C rows.                                   |
10                       | Matrix C columns.                                |
11                       | Internal mode word.                              |
12                       | Last status byte.                                |
13                       | Clock ticks per millisecond.                     |
14                       | Current length of string buffer.                 |
15                       | String selection starting point.                 |
16                       | String selection length.                         |
17                       | 8-bit character at string selection point.       |
18                       | Number of bytes in instruction buffer.           |

Examples

```plaintext
value = READVAR(15); returns the start of the string selection point
```

See Also

*um-FPU64 Instruction Set: READVAR*
**RETURN**

Returns from a user-defined procedure or function.  
*Note:* Must be used inside a user-defined procedure or function.

**Syntax**

```
RETURN [returnValue]
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>returnValue</td>
<td>long expression</td>
<td>The value returned from a user-defined function.</td>
</tr>
<tr>
<td></td>
<td>float expression</td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

User-defined procedure have no return value. User-defined functions must return a value.

**Examples**

```
#function 1getID() long
  return 35 ; return the value 35
#end
```

**See Also**

CONTINUE, DO...WHILE...UNTIL...LOOP, EXIT, FOR...NEXT, IF...THEN
RTC

Manage the real-time clock.

Syntax

```
RTC(INIT[+RTCC][+ALARM_OUT][+HZ_OUT][+CAL][+ALARM_ON])
RTC(START)
RTC(STOP)
RTC(ALARM_MASK[+mask])
RTC(WRITE_TIME[+DATE_TIME][+MONTH][+DAY][+HOUR][+MINUTE][+SECOND][+WEEKDAY][,dateTime])
RTC(WRITE_TIME+STR[+DATE_TIME][+YEAR][+MONTH][+DAY][+HOUR][+MINUTE][+SECOND][+WEEKDAY][,string])
RTC(WRITE_ALARM[+DATE_TIME][+YEAR][+MONTH][+DAY][+HOUR][+MINUTE][+SECOND][+WEEKDAY][,dateTime])
RTC(WRITE_ALARM+STR[+DATE_TIME][+YEAR][+MONTH][+DAY][+HOUR][+MINUTE][+SECOND][+WEEKDAY][,string])
RTC(READ_TIME[+DATE_TIME][+MONTH][+DAY][+HOUR][+MINUTE][+SECOND][+WEEKDAY][,dateTime])
RTC(READ_TIME+STR[+DATE_TIME][+YEAR][+MONTH][+DAY][+HOUR][+MINUTE][+SECOND][+WEEKDAY][,string])
result = RTC(READ_TIME)
RTC(READ_ALARM[+DATE_TIME][+MONTH][+DAY][+HOUR][+MINUTE][+SECOND][+WEEKDAY][,dateTime])
RTC(READ_ALARM+STR[+DATE_TIME][+YEAR][+MONTH][+DAY][+HOUR][+MINUTE][+SECOND][+WEEKDAY][,string])
result = RTC(READ_ALARM)
RTC(NUM_TO_STR[+DATE_TIME][+DATE][+TIME][,dateTime])
RTC(STR_TO_NUM[+DATE_TIME][+DATE][+TIME][,string])
RTC(NUM_TO_DATE, register[, dateTime])
RTC(DATE_TO_NUM, register)
result = RTC(STR_TO_NUM)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>pre-defined symbols</td>
<td>The real-time clock action and modifiers.</td>
</tr>
<tr>
<td>dateTime</td>
<td>long expression</td>
<td>Numeric date and time value. If no expression is included the current value in register 0 will be used.</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td>Date and time string. If no string is included the current contents of the string buffer will be used.</td>
</tr>
</tbody>
</table>

Notes

See the uM-FPU64 Instruction Set document for detailed descriptions of the RTC actions.

Examples

```
RTC(WRITE_TIME, "2010-08-11 14:30:00") ; write RTC date and time
```

See Also

uM-FPU64 Instruction Set: RTC
SAVEMA
SAVEMB
SAVEMC

Store a matrix value.

Syntax

\[
\begin{align*}
\text{SAVEMA}(\text{row}, \text{column}, \text{value}) \\
\text{SAVEMB}(\text{row}, \text{column}, \text{value}) \\
\text{SAVEMC}(\text{row}, \text{column}, \text{value})
\end{align*}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>row</td>
<td>long constant</td>
<td>The row number of the matrix element, or a register containing the row number.</td>
</tr>
<tr>
<td>column</td>
<td>long constant</td>
<td>The column number of the matrix element, or a register containing the column number.</td>
</tr>
<tr>
<td>value</td>
<td>float expression</td>
<td>The value to store at the specified row and column.</td>
</tr>
</tbody>
</table>

Notes
These procedures store a value at the specified row and column of a matrix. The row and column numbers start from zero. If the row or column values are out of range, no value is stored.

Examples

\[
\begin{align*}
&\text{SELECTMA}(100, 3, 3) &;\text{ matrix } A \text{ is defined as a 3x3 matrix starting at register 100} \\
&MOP(\text{SCALAR\_SET}, 0) &;\text{ set all values in matrix } A \text{ to zero} \\
&\text{SAVEMA}(0, 2, \pi) &;\text{ store the value } \pi \text{ at row 0, column 2}
\end{align*}
\]

See Also
MOP, LOADMA, LOADMB, LOADMC, SELECTMA, SELECTMB, SELECTMC

\textit{uM-FPU64 Instruction Set: SAVEMA, SAVEMB, SAVEMC}
SELECT...CASE

Executes one of a group of statements, depending on the value of the expression or string.  
*Note:* Must be used inside a user-defined procedure or function.

**Syntax**

```
SELECT compareItem

[CASE compareValue [, compareValue]...  
  statements]...

[ELSE  
  statements]
ENDSELECT
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compareItem</td>
<td>A numeric expression or string procedure.</td>
</tr>
<tr>
<td>compareValue</td>
<td>A numeric or string constant.</td>
</tr>
<tr>
<td>statements</td>
<td>One or more statements that execute if a compareValue is equal to the value of compareItem.</td>
</tr>
</tbody>
</table>

**Notes**

The SELECT clause specifies a numeric expression or string procedure that will be used in the CASE clauses. If a numeric expression is specified, then all compareValues in the CASE clauses must be a numeric constants of the same data type as the compareItem. If the STRSEL or STRFIELD procedure is specified, then all compareValues in the CASE clauses must be a string constants. The CASE clauses are evaluated sequentially. If a compareValue is equal to the compareItem, the statements in that CASE clause are executed. If no CASE clause has a match and an ELSE clause is included, the statements in the ELSE clause are executed.

**Examples**

```
n  equ  L10
SELECT n

CASE 1
  strset("Blue") ; if n = 1, then set string = Blue,

CASE 2, 3
  strset("Green") ; if n = 2 or n = 3, then set string = Green

ELSE
  strset("Black") ; otherwise, set string = Black

ENDSELECT
```
n equ L10

SELECT STRSEL(0,127) ; select entire string buffer for comparison

CASE "Blue"
   n = 1 ; if string = Blue, then set n = 1

CASE "Green", "Red"
   n = 2 ; if string=Green or string = Red, then set n = 2

ELSE
   n = 0 ; otherwise, set n = 0

ENDSELECT

See Also
DO...WHILE...UNTIL...LOOP, FOR...NEXT, IF...THEN, IF...THEN...ELSE
**SELECTA**

Select register A.

**Syntax**

```plaintext
SELECTA(register)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>register</td>
<td>The register to select as register A.</td>
</tr>
</tbody>
</table>

**Notes**

This procedure is rarely required, since the compiler selects register A automatically during code generation.

**Examples**

```plaintext
SELECTA(F100) ; select register 100 as register A
SELECTA(L1)  ; select register 1 as register A
```

**See Also**

SELECTX

*uM-FPU64 Instruction Set: SELECTA*
**SELECTMA**  
**SELECTMB**  
**SELECTMC**

Select the registers used for matrix operations.

**Syntax**

```plaintext
SELECTMA(register|pointer, rows, columns)
SELECTMB(register|pointer, rows, columns)
SELECTMC(register|pointer, rows, columns)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>register</td>
<td>The first register of the matrix.</td>
</tr>
<tr>
<td>pointer</td>
<td>pointer</td>
<td>Pointer to first element of the matrix.</td>
</tr>
<tr>
<td>row</td>
<td>long constant</td>
<td>The number of rows, or a register containing the number of rows.</td>
</tr>
<tr>
<td>column</td>
<td>long constant</td>
<td>The number of columns, or a register containing the number of columns.</td>
</tr>
</tbody>
</table>

**Notes**

The `register` parameter specifies the first register of the matrix. The `pointer` parameter points to the first element of a matrix. The `rows` and `columns` parameters specify the size of the matrix. Matrix values are stored in sequential registers. Register X is also set to point to the first register of the matrix.

**Examples**

```plaintext
SELECTMA(F100, 3, 3) ; matrix A is defined as a 3x3 matrix starting at register 100
SELECTMB(F109, 2, 3) ; matrix B is defined as a 2x3 matrix starting at register 109
SELECTMC(F115, 3, 1) ; matrix C is defined as a 3x1 matrix starting at register 115
```

**See Also**

- MOP, LOADMA, LOADMB, LOADMC, SAVEMA, SAVEMB, SAVEMC
- uM-FPU64 Instruction Set: SELECTMA, SELECTMB, SELECTMC
SELECTX
Select register X.

Syntax
SELECTA(register)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>register</td>
<td>The register to select as register X.</td>
</tr>
</tbody>
</table>

Notes
This procedure is used to set register X prior to using the \([X]\) operator.

Examples

```
SELECTX(F20) ; set register X to point to register 20
[X] = 10.5   ; stores 10.5 to the register pointed to by register X, then increments X
```

See Also
SELECTA

uM-FPU64 Instruction Set: SELECTX
SERIAL

The SERIAL function and procedures are used to send serial data to the SEROUT pin and read serial data from the SERIN pin. The first argument of the SERIAL function or procedure is a special symbol name that identifies the type of operation. The SERIAL operations are summarized as follows:

- SERIAL(SET_BAUD, baud)
- SERIAL(WRITE_STR, string)
- SERIAL(WRITE_STRZ, string)
- SERIAL(WRITE_SBUF)
- SERIAL(WRITE_SSEL)
- SERIAL(WRITE_CHAR, value)
- SERIAL(WRITE_FLOAT, value, format)
- SERIAL(WRITE_LONG, value, format)
- SERIAL(WRITE_COMMA)
- SERIAL(WRITE_CRLF)
- SERIAL(DISABLE_INPUT)
- SERIAL(ENABLE_CHAR)
- SERIAL(STATUS_CHAR)
  result = SERIAL(READ_CHAR)
- SERIAL(ENABLE_NMEA)
- SERIAL(STATUS_NMEA)
- SERIAL(READ_NMEA)

See Also

uM-FPU64 Instruction Set: SEROUT, SERIN

A detailed description of each SERIAL operation is shown below.

Syntax

SERIAL(SET_BAUD, baud)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>baud</td>
<td>long constant</td>
<td>The baud rate for the SEROUT and SERIN pins.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or 115200)</td>
</tr>
</tbody>
</table>

Notes

Sets the baud rate for both the SEROUT and SERIN pins. If the baud rate is specified as 0, the FPU debug mode is enabled and the baud rate is set to 57,600 baud. For all other baud rates, the FPU debug mode is disabled, so the SEROUT and SERIN pins can be used for serial data transfers.

Examples

SERIAL(SET_BAUD, 4800) ; sets the baud rate to 4800 baud

Syntax

SERIAL(WRITE_STR, string)
**String Constants**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string constant</td>
<td>The string to send to the specified serial output.</td>
</tr>
</tbody>
</table>

**Notes**

Writes the string to the specified serial output.

**Examples**

```
SERIAL(WRITE_STR, "abc") ; sends abc to the SEROUT pin
```

**Syntax**

```
SERIAL(WRITE_STRZ, string)
```

**Notes**

Writes the string to the specified serial output, followed by a zero byte.

**Examples**

```
SERIAL(WRITE_STRZ, "abc") ; sends abc and a zero byte to the SEROUT pin
```

**Syntax**

```
SERIAL(WRITE_SBUF)
```

**Notes**

Writes the contents of the string buffer to the specified serial output.

**Examples**

```
SERIAL(WRITE_STRBUF) ; sends contents of the string buffer to the SEROUT pin
```

**Syntax**

```
SERIAL(WRITE_STRSEL)
```

**Notes**

Writes the current string selection to the specified serial output.

**Examples**

```
SERIAL(WRITE_STRSEL) ; sends current string selection to the SEROUT pin
```
Syntax

\texttt{SERIAL(WRITE\_CHAR, value)}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>long expression</td>
<td>The lower 8 bits are output to the specified serial output.</td>
</tr>
</tbody>
</table>

Notes

Writes the lower 8 bits of the value to the specified serial output.

Examples

\begin{verbatim}
SERIAL(WRITE\_CHAR, $32) ; sends $32 (the digit 2) to the SEROUT pin
SERIAL(WRITE\_CHAR+ASYNC, value) ; sends the lower 8 bits of value to the ASYNC pin
\end{verbatim}

Syntax

\texttt{SERIAL(WRITE\_FLOAT, value, format)}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>float expression</td>
<td>The floating point value to convert.</td>
</tr>
</tbody>
</table>

Notes

Converts the \texttt{value} to a floating point string with the specified \texttt{format}, and writes the string to the specified serial output.

Examples

\begin{verbatim}
SERIAL(WRITE\_FLOAT, pi, 42) ; sends 3.14 to the SEROUT pin
SERIAL(WRITE\_FLOAT, value, 0) ; sends floating point string to the SEROUT pin
\end{verbatim}

Syntax

\texttt{SERIAL(WRITE\_LONG, value, format)}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>long expression</td>
<td>The long integer value to convert.</td>
</tr>
</tbody>
</table>

Notes

Converts the \texttt{value} to a floating point string with the specified \texttt{format}, and writes the string to the specified serial output.

Examples

\begin{verbatim}
SERIAL(WRITE\_FLOAT, pi, 42) ; sends 3.14 to the serial output
SERIAL(WRITE\_FLOAT, value, 0) ; sends floating point string to the SEROUT pin
\end{verbatim}
Syntax

**SERIAL**(WRITE_COMMA)

**Notes**
Writes a comma to the specified serial output.

**Examples**

```plaintext
SERIAL(WRITE_COMMA) ; sends comma to the SEROUT pin
```

Syntax

**SERIAL**(WRITE_CRLF)

**Notes**
Writes a carriage return and linefeed to the specified serial output.

**Examples**

```plaintext
SERIAL(WRITE_CRLF) ; sends CR/LF to the SEROUT pin
```

Syntax

**SERIAL**(DISABLE_INPUT)

**Notes**
The SERIN pin is disabled.

**Examples**

```plaintext
SERIAL(DISABLE_INPUT) ; disables the SERIN pin
```

Syntax

**SERIAL**(ENABLE_CHAR)

**Notes**
The SERIN or ASYNC pin is enabled for character input. Received characters are stored in a 160 byte input buffer. The serial input status can be checked with the **SERIAL**(STATUS_CHAR) procedure and characters can be read using the **SERIAL**(READ_CHAR) function.

**Examples**

```plaintext
SERIAL(ENABLE_CHAR) ; enables the SERIN or ASYNC pin for character input
```
Syntax

```
SERIAL(STATUS_CHAR)
```

Notes
The FPU status byte is set to zero (Z) if the character input buffer is empty, or non-zero (NZ) if the input buffer is not empty.

Examples

```
SERIAL(STATUS_CHAR) ; get the character input status
if STATUS(Z) then return ; return from the function if the buffer is empty
```

Syntax

```
result = SERIAL(READ_CHAR)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The next available serial character value.</td>
</tr>
</tbody>
</table>

Notes
Wait for the next available serial input character, and return the character. This function only waits if the instruction buffer is empty. The IDE compiler automatically adds an FPU wait call if the function is called from microcontroller code. If this function is used in a user-defined function, the user must be sure that an FPU wait call is inserted in the microcontroller code immediately after the user-defined function call. If there are other instructions in the instruction buffer, or another instruction is sent before the `SERIAL(READ_CHAR)` function has completed, it will terminate and return a zero value.

Examples

```
ch = SERIAL(READ_CHAR) ; returns the next serial input character from SERIN
ch = SERIAL(READ_CHAR+ASYNC) ; returns the next serial input character from ASYNC
```

Syntax

```
SERIAL(ENABLE_NMEA)
```

Notes
The SERIN or ASYNC pin is enabled for NMEA input. Serial input is scanned for NMEA sentences which are then stored in a 200 byte buffer. This allows subsequent NMEA sentences to be buffered while the current sentence is being processed. The sentence prefix character ($), trailing checksum characters (if specified), and the terminator (CR, LF) are not stored in the buffer. NMEA sentences are transferred to the string buffer for processing using the `SERIAL(READ_NMEA)` procedure, and the NMEA input status can be checked with the `SERIAL(STATUS_NMEA)` procedure.

Examples

```
SERIAL(ENABLE_NMEA) ; enables the SERIN pin for NMEA input
SERIAL(ENABLE_NMEA+ASYNC) ; enables the ASYNC pin for NMEA input
```
Syntax

**SERIAL**(STATUS_NMEA)

Notes

The FPU status byte is set to zero (Z) if the NMEA sentence buffer is empty, or non-zero (NZ) if at least one NMEA sentence is available in the buffer.

Examples

```plaintext
SERIAL(STATUS_NMEA) ; get the NMEA input status
if STATUS(Z) then return ; return from the function if the buffer is empty
```

Syntax

**SERIAL**(READ_NMEA)

Notes

Read the next NMEA sentence from the NMEA input buffer and transfer it to string buffer. The first field of the string is automatically selected so that the **STRCMP** function can be used to check the sentence type. If the sentence is valid, the FPU status byte is set to greater-than (GT). If an error occurred, the FPU status byte is set to less-than (LT) and the special status bits NMEA_CHECKSUM and NMEA_OVERRUN are set. The **STATUS** function can be used to check these bits. This procedure only waits if the instruction buffer is empty. The IDE compiler automatically adds an FPU wait call if the procedure is called from microcontroller code. If this procedure is used in a user-defined function, the user must be sure that an FPU wait call is inserted in the microcontroller code immediately after the function call. If there are other instructions in the instruction buffer, or another instruction is sent before the **SERIAL**(READ_NMEA) procedure has completed, it will terminate and the string buffer will be empty.

Examples

```plaintext
SERIAL(READ_NMEA) ; sends abc to the SEROUT pin
if STATUS(GT) then return ; return from
```
SETIND

Return a pointer to a register or memory location.

Syntax

\[
\begin{align*}
\text{SETIND} & (\text{type}, \text{reg}) \\
\text{SETIND} & (\text{type}, \text{address}) \\
\text{SETIND} & (\text{type}, \text{function}, \text{offset})
\end{align*}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>symbol</td>
<td>Specifies data type of the pointer.</td>
</tr>
<tr>
<td></td>
<td>REG_LONG,</td>
<td>REG_FLOAT, FLASH_UINT8, FLASH_INT16, MEM_UINT16, FLASH_LONG32, FLASH_FLOAT32,</td>
</tr>
<tr>
<td></td>
<td>MEM_UINT16,FLASH_LONG64, FLASH_FLOAT64, DMA_UINT8,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DMA_INT16, DMA_UINT16, DMA_LONG32,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DMA_FLOAT32, DMA_LONG64, DMA_FLOAT64,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLASH_UINT8, FLASH_INT16, FLASH_UINT16,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLASH_LONG32, FLASH_FLOAT32, FLASH_LONG64,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLASH_FLOAT64</td>
<td></td>
</tr>
<tr>
<td>reg</td>
<td>register</td>
<td>The register number for the pointer. Required for REG data types.</td>
</tr>
<tr>
<td>address</td>
<td>long constant</td>
<td>The memory address for the pointer. Required for MEM and DMA data types.</td>
</tr>
<tr>
<td>function</td>
<td>function</td>
<td>The function number for the pointer. Required for FLASH data types.</td>
</tr>
<tr>
<td>offset</td>
<td>long constant</td>
<td>The function offset for the pointer. Required for FLASH data types.</td>
</tr>
</tbody>
</table>

Notes

This function is used to set a pointer value. The left side of the equation must be a pointer.

Examples

\[
\begin{align*}
p & = \text{SETIND}(\text{REG_FLOAT}, \text{F10}) \quad ; \text{sets pointer to register 10, data type is Float} \\
p & = \text{SETIND}(\text{MEM_INT8}, 100) \quad ; \text{sets pointer to RAM address 100, data type is int8} \\
p & = \text{SETIND}(\text{FLASH_FLOAT32}, 0, 0) \quad ; \text{sets pointer to Flash function 0, offset 0}
\end{align*}
\]

See Also

COPYIND

uM-FPU64 Instruction Set: SETIND
STATUS
Checks the FPU status bits.

Syntax
STATUS(conditionCode)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conditionCode</td>
<td>literal string</td>
<td>A condition code symbol.</td>
</tr>
</tbody>
</table>

Notes
This function can only be used in a conditional expression. The STATUS condition is true if the FPU status byte agrees with if the specified conditionCode. If the NMEA_CHECKSUM or NMEA_OVERRUN conditionCode is specified, the STATUS condition is true if the corresponding bit is set.

Examples

```plaintext
if status(LT) then
  if status(NMEA_OVERRUN) then
    return -1
  elseif status(NMEA_CHECKSUM) then
    return -2
  endif
endif
```

See Also
conditional expression
**STRBYTE**

Insert 8-bit character at the string selection point.

**Syntax**

```
STRBYTE(value)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>long expression</td>
<td>8-bit character to insert</td>
</tr>
</tbody>
</table>

**Notes**

The 8-bit character is stored at the string selection point. If the selection length is zero, the 8-bit character is inserted into the string at the selection point. If the selection length is not zero, the selected characters are replaced. The selection point is updated to point immediately after the inserted string, so multiple insertions can be appended.

**Examples**

Note: In the following example, {} characters are used to shown the string selection point.

```
n  equ  L10

STRSET("");  string buffer = {}
n = 36
STRBYTE(0x30+n/10); stores the digit 3 (0x33), string buffer = 3{}
STRBYTE(0x30+n%10); stores the digit 6 (0x36), string buffer = 36{}
```

**See Also**

FTOA, LTOA, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSEL, STRSET

_uM-FPU64 Instruction Set: STRBYTE_
STRFCHR

Sets the field separator characters used by the STRFIELD procedure.

Syntax

\[ \text{STRFCHR}(\text{string}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>A string containing the list of field separator characters.</td>
</tr>
</tbody>
</table>

Notes

The default field separator is a comma. This procedure can be used to select other field separators. The order of the characters in the \textit{string} is not important.

Examples

See the examples for STRFIELD.

See Also

FTOA, LTOA, STRBYTE, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSEL, STRSET

\textit{uM-FPU64 Instruction Set}: STRINC, STRDEC
**STRFIELD**

Find the specified field in the string.

**Syntax**

\[
\text{STRFIELD}([field])
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td>register</td>
<td>Specifies the field number.</td>
</tr>
<tr>
<td></td>
<td>long constant</td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

The `field` parameter can be a register or a long constant. If a register is specified, the value of the register specifies the field number. Fields are numbered from 1 to n, and are separated by the field separator characters. The default field separator character is the comma. Other field separators can be specified using the `STRFCHR` procedure. The selection point is set to the specified field. If the field number is zero, the selection point is set to the start of the buffer. If the field number is greater than the number of fields, the selection point is set to the end of the buffer. `STRFIELD` can also be used in a conditional expression.

**Examples**

The following example shows how a date/time string can be parsed.

Note: In the following example the `{}` characters are used to shown the string selection point.

```assembly
year    equ  L10
minute  equ  L11

STRSET("2010-7-20 10:57 pm") ; string buffer = 2010-7-20 10:57 pm{}
STRFCHR(":-: ") ; use dash, colon, space as field separated
STRFIELD(1) ; string buffer = {2010}-7-20 10:57 pm
year = STRLONG() ; convert string to year
minutes = STRLONG() ; convert string to minutes

if strfield() = "GPRMC" then ; check for GPRMC sentence
    ...
endif
```

**See Also**

FTOA, LTOA, STRBYTE, STRFCHR, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSEL, STRSET

*uM-FPU64 Instruction Set: STRINC, STRDEC*
STRFIND

Find the string in the current string selection.

Syntax

\[ \text{STRFIND}(\text{string}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>The string to find in the string selection.</td>
</tr>
</tbody>
</table>

Notes

This procedure searches in the current string selection for the specified \textit{string}. If the \textit{string} is found, the string selection is changed to select the matching string.

Examples

Note: In the following example the \{\} characters are used to shown the string selection point.

\[
\begin{align*}
\text{STRSET}(\text{"abcdef"}) & \quad ; \text{string buffer} = \text{abcdef}\{} \\
\text{STRSEL}(0,127) & \quad ; \text{string buffer} = \{\text{abcdef}\} \\
\text{STRFIND}(\text{"d"}) & \quad ; \text{string buffer} = \text{abc}\{\text{d}\}\text{ef}
\end{align*}
\]

See Also

FTOA, LTOA, STRBYTE, STRFCHR, STRFIELD, STRFLOAT, STRINC, STRINS, STRLONG, STRSEL, STRSET

\textit{uM-FPU64 Instruction Set: STRINC, STRDEC}
**STRFLOAT**

Returns the floating point value of the current string selection.

**Syntax**

```plaintext
result = STRFLOAT()
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>float</td>
<td>The converted value.</td>
</tr>
</tbody>
</table>

**Notes**

Converts the current string selection to a floating point value, and returns the `result`. Conversion stops at the first character that is not a valid character for a floating point number.

**Examples**

Note: In the following example the {} characters are used to shown the string selection point.

```plaintext
; assume string buffer = 35.5,1e5,100{}
STRSEL(5,7) ; string buffer = 35.5,{1e5,100}
result = STRFLOAT() ; returns 100000.0 (terminates on the comma)
STRSEL(0,255) ; string buffer = {35.5,1e5,100}
result = STRFLOAT() ; returns 35.5 (terminates on the comma)
```

**See Also**

`STRBYTE`, `STRFCHR`, `STRFIELD`, `STRFIND`, `STRINC`, `STRINS`, `STRLONG`, `STRSEL`, `STRSET`, `FTOA`, `LTOA`

*uM-FPU64 Instruction Set: STRTOF*
**STRINC**

Increment or decrement the string selection point.

**Syntax**

```plaintext
STRINC(increment)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>increment</td>
<td>register</td>
<td>Specifies the increment or decrement amount.</td>
</tr>
<tr>
<td></td>
<td>long constant</td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

The `increment` parameter can be a register or a long constant. If a register is specified, the value of the register specifies the increment or decrement value. If the value is positive, the selection point is incremented. If the value is negative, then selection point is decremented.

**Examples**

Note: In the following example the {} characters are used to shown the string selection point.

```plaintext
n  equ  L10

STRSET("abcdef") ; string buffer = abcdef{}
STRSEL(0,127)    ; string buffer = {abcdef}
STRFIND("d")    ; string buffer = abc{d}ef
STRINC(-2)       ; string buffer = a{}bcd{ef
STRINS("x")     ; string buffer = ax{}bcdef
n  = 3
STRINC(n)        ; string buffer = axbcd{}ef
STRINS("y")     ; string buffer = axbcdy{}ef
```

**See Also**

FTOA, LTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINS, STRLONG, STRSEL, STRSET

*uM-FPU64 Instruction Set: STRINC, STRDEC*
STRINS
Insert string at the string selection point.

Syntax
`STRINS(string)`

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>String to insert at selection point.</td>
</tr>
</tbody>
</table>

Notes
The `string` is stored at the string selection point. If the selection length is zero, the `string` is inserted at the selection point. If the selection length is not zero, the selected characters are replaced. The selection point is updated to point immediately after the inserted string, so multiple insertions can be appended.

Examples
Note: In the following example the {} characters are used to shown the string selection point.

```
STRSET("abcd") ; string buffer = abcd{}
STRSEL(1,2)     ; string selection = a{bc}d
STRINS("x")   ; string buffer = ax{}
STRINS("yz")  ; string buffer = axy{}zd
```

See Also
- FTOA, LTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS,
- STRLONG, STRSEL, STRSET
- uM-FPU64 Instruction Set: STRINC
**STRLONG**

Returns the long integer value of the current string selection.

**Syntax**

```
result = STRLONG()
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The converted value.</td>
</tr>
</tbody>
</table>

**Notes**

Converts the current string selection to a long integer value, and returns the `result`. Conversion stops at the first character that is not a valid character for a long integer number.

**Examples**

Note: In the following example, `{}` characters are used to shown the string selection point.

```
; assume string buffer = 35.5,1e5,100{}
STRSEL(5,7) ; string buffer = 35.5,{1e5,100}
result = STRFLOAT() ; returns 1 (terminates on the e)
STRSEL(0,255) ; string buffer = {35.5,1e5,100}
result = STRFLOAT() ; returns 35 (terminates on the decimal point)
```

**See Also**

`STRBYTE`, `STRCHR`, `STRFIELD`, `STRFIND`, `STRFLOAT`, `STRING`, `STRINS`, `STRSET`, `FTOA`, `LTOA`  
`uM-FPU64 Instruction Set: STRTOL`
String Constant

A string constant is enclosed in double quote characters. Special characters can be entered using a backslash prefix. The special characters are as follows:

- \r: carriage return (0x0D)
- \n: linefeed (0x0A)
- \t: horizontal tab (0x09)
- \v: vertical tab (0x0B)
- \\
: backslash
- ": double quote
- \xx: 8-bit value (where xx are hexadecimal digits, e.g. \0C"

Examples

<table>
<thead>
<tr>
<th>String Constant</th>
<th>Actual String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GPRMC&quot;</td>
<td>GPRMC</td>
</tr>
<tr>
<td>&quot;N&quot;</td>
<td>N</td>
</tr>
<tr>
<td>&quot;sample&quot;</td>
<td>sample</td>
</tr>
<tr>
<td>&quot;string2\0D\0A&quot;</td>
<td>string2&lt;carriage return&gt;&lt;linefeed&gt;</td>
</tr>
<tr>
<td>&quot;5\3&quot;</td>
<td>5\3</td>
</tr>
<tr>
<td>&quot;this &quot;one&quot;&quot;</td>
<td>this &quot;one&quot;</td>
</tr>
</tbody>
</table>
STRSEL
Set the string selection point

Syntax
STRSEL([start,] length)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>register or long constant</td>
<td>The start of the string selection.</td>
</tr>
<tr>
<td>length</td>
<td>long expression</td>
<td>The length of the string selection.</td>
</tr>
</tbody>
</table>

Notes
If the start parameter is not specified, the start of the current string selection is used. The start parameter can be a register or a long constant. If a register is specified, the value of the register specifies the start of the selection point. If the start value is greater than the length of the string buffer, it is adjusted to the end of the buffer. The length parameter can be any long expression. If the string selection exceeds the length of the string buffer, it is adjusted to fit the string buffer. STRSEL can also be used in a conditional expression.

Examples
Note: In the following example, {} characters are used to shown the string selection point.

```
n equ L10
STRSET("0123456789ABCDEF") ; string buffer = 0123456789ABCDEF{}
STRSEL(5, 3) ; string buffer = 01234{567}89ABCDEF
n = 11
STRSEL(n, 1) ; string buffer = 0123456789A{B}CDEF
if STRSEL(2,2) = “W1” then ; check if selection = “W1”
  ...
endif
```

See Also
FTOA, LTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINGC, STRINS, STRLONG, STRSET

uM-FPU64 Instruction Set: STRINGC, STRDEC
STRSET

Copy the string to the string buffer.

Syntax

\texttt{STRSET}(\textit{string})

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{string}</td>
<td>string</td>
<td>String to store in string buffer.</td>
</tr>
</tbody>
</table>

Notes

The \textit{string} is stored in the string buffer, and the selection point is set to the end of the string buffer.

Examples

Note: In the following example the \{\} characters are used to shown the string selection point.

\texttt{STRSET(“abcd”) ; string buffer = abcd{} }

See Also

FTOA, LTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSEL

\textit{uM-FPU64 Instruction Set: STRSET}
TICKLONG

Returns the number of milliseconds or microseconds that have elapsed since the FPU timer was started. Milliseconds are returned if a 32-bit register is selected. Microseconds are returned if a 64-bit register is selected.

Syntax

\[ \text{result} = \text{TICKLONG}() \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The number of milliseconds or microseconds since the FPU timer was started.</td>
</tr>
</tbody>
</table>

Notes

Returns the number of milliseconds that have elapsed since the FPU timer was started by the TIMESET procedure. The internal millisecond timer is a 32-bit register.

Examples

\[ \text{result} = \text{TICKLONG}() \]; returns the number of msec since the FPU timer was started

See Also

TIMELONG, TIMESET

uM-FPU64 Instruction Set: TICKLONG
TIMELONG
Returns the number of seconds that have elapsed since the FPU timer was started.

Syntax
\[ result = \text{TIMELONG}() \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The number of seconds since the FPU timer was started.</td>
</tr>
</tbody>
</table>

Notes
Returns the number of seconds that have elapsed since the FPU timer was started by the TIMESET procedure. The internal seconds timer is a 32-bit register.

Examples
\[
\text{result} = \text{TIMELONG}() \quad ; \text{returns the number of seconds since the FPU timer was started}
\]

See Also
TICKLONG, TIMESET
uM-FPU64 Instruction Set: TIMELONG
TIMESET

Set internal timer values.

Syntax

\texttt{TIMESET(\textit{seconds})}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{seconds}</td>
<td>long expression</td>
<td>The internal seconds timer is set to this value.</td>
</tr>
</tbody>
</table>

Notes

The internal seconds timer is set to \textit{seconds} and the internal millisecond timer is set to zero.

Examples

\texttt{TIMESET(0) ; set seconds timer and msec timer to zero}

See Also

\texttt{TICKLONG, TIMELONG}

\textit{uM-FPU64 Instruction Set: TIMESET}
TRACEON, TRACEOFF

Turn the debug instruction trace on or off.

Syntax

TRACEON
TRACEOFF

Notes

These procedures provide manual control over the debug instruction trace. They can be used to only trace specific sections of code. If the debugger is disabled, these procedures are ignored.

Examples

```
TRACEON ; turn on debug trace
 ; all instructions in this section are traced
TRACEOFF ; turn off debug trace
 ; no instructions in this section are traced
TRACEON ; turn on debug trace
```

See Also

TRACEREG, TRACESTR, BREAK

uM-FPU64 Instruction Set: TRACEON, TRACEOFF
**TRACEREG**
Display register value in the debug trace.

**Syntax**

```
TRACEREG(register)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>register</td>
<td>Register to trace.</td>
</tr>
</tbody>
</table>

**Notes**
If the debugger is enabled, the register number, hexadecimal value, long integer value, and the floating point value of the `register` contents are displayed in the debug window. If the debugger is disabled, this procedure is ignored.

**Examples**
In this example, the following text would be displayed in the debug trace window.

```
R10:00000005, 5, 7.006492e-45
R11:3FC00000, 1069547520, 1.5
```

```
cnt   equ L10
value equ F11

cnt = 5  ; set long integer value
value = 1.5 ; set floating point value
TRACEREG(cnt) ; displays register 10 in debug trace
TRACEREG(value) ; displays register 11 in debug trace
```

**See Also**
BREAK, TRACEOFF, TRACEON, TRACESTR

*uM-FPU64 Instruction Set: TRACEREG*
**TRACESTR**

Display message string in the debug trace.

**Syntax**

```
TRACESTR(string)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>The message string.</td>
</tr>
</tbody>
</table>

**Notes**

If the debugger is enabled, the `string` is displayed in the debug trace window. If the debugger is disabled, this procedure is ignored.

**Examples**

In this example, the following text would be displayed in the debug trace window.

```
"test1"
```

```
TRACESTR("test1") ; display trace message in debug trace
```

**See Also**

BREAK, TRACEOFF, TRACEON, TRACEREG

*uM-FPU64 Instruction Set: TRACESTR*
**User-defined Functions**

User-defined functions can be stored in Flash memory on the uM-FPU64 chip.

**Defining Functions**

The `#FUNCTION` directive are used to define Flash memory functions. All statements between the `#FUNCTION` and the next `#FUNCTION` or `#END` directive will be compiled and stored as part of the function.

The `#FUNC` directive can be used at the start of the program to define functions prototypes. The use of function prototypes is recommended. It allows the allocation of function storage to be easily maintained, and supports calling functions that are defined later in the program.

Functions can optionally define parameters to be passed when the function is called, and can optionally return a value. A procedure is a function with no return value. The data type of the parameters and the return value must be declared when the function is declared. The data types are as follows:

- **FLOAT** 32-bit floating point
- **LONG** 32-bit long integer
- **ULONG** 32-bit unsigned long integer
- **FLOAT64** 64-bit floating point
- **LONG64** 64-bit long integer
- **ULONG64** 64-bit unsigned long integer

**Passing Parameters and Return Values**

When parameters are defined for a function, the parameter values are passed in registers 1 through 9, with the first parameter in register 1, the second parameter in register 2, etc. The compiler automatically defines local symbols `arg1`, `arg2`, ... with the correct data type. These symbols can then be used inside the function. When a return value is defined for a function, the value specified by a return statement is returned by the function in register 0. A `RETURN` statement must be the last statement of all functions that return a value.

**Calling Functions**

Once a function has been defined using a `#FUNC` or `#FUNCTION` directive, the function can be called simply by using the function name in a statement or expression. Functions (user-defined functions that return a value) can be used in expressions. Procedures (user-defined functions that don’t return a value) are called as a
statement. If a function has no arguments, a set a parenthesis is still required. If a procedure has no arguments, the parentheses are optional.

<table>
<thead>
<tr>
<th>n = getID()</th>
<th>; function call</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = y + addOffset(y)</td>
<td>; function call</td>
</tr>
<tr>
<td>update</td>
<td>; procedure call</td>
</tr>
<tr>
<td>getLocation(1)</td>
<td>; procedure call</td>
</tr>
</tbody>
</table>

**Nested Functions Calls**

Functions can call other functions, with a maximum of 16 levels of nesting supported. Since all function parameters are passed in registers 1 to 9, care must be taken to ensure that the value of registers 1 to 9 are still valid after a nested function call. The values passed as arg1, arg2, ... may be modified by calling another function. If parameter values need to be used after other nested function calls, they should be copied to other registers first.

**See Also**

#END, #FUNC, #FUNCTION

*uM-FPU64 Instruction Set: FCALL, RET, RET,cc*
**XOP (extended opcode) Instructions**

XOP (extended opcode) instructions can be loaded from XOP library files and stored in Flash memory on the uM-FPU64 chip.

**Defining XOPs**

XOP instructions are loaded into Flash memory as required by the program. Each XOP used in a program must have an #XOP directive specified previously in the program to load the XOP code, and define the arguments and return value (if any).

```plaintext
#XOP quaternion:q_add ; loads the q_add XOP instruction
#XOP quaternion:q_norm ; loads the q_norm XOP instruction
```

**Passing Arguments to the Quaternion XOPs**

XOP instructions are called in a similar manner to calling a procedure or function, but the argument passing method is different. XOP instructions can have up to three arguments. The arguments are passed as 8-bit bytes immediately following the XOP opcode. The arguments refer to 32-bit registers or 64-bit registers as defined by the XOP. If bit 7 of the byte is 0, then bits 6:0 contain the register number. If bit 7 of the byte is 1, then bits 6:0 contain the register number of a register containing a pointer a register. The uM-FPU64 IDE takes care of assigning the correct bit values based on the datatype of the argument.

```plaintext
q_add(qa, qb, qc) ; add quaternion add XOP
tmp = q_norm(qa) ; calculate the norm of the quaternion
```
#ASM
Start of assembler code.

Syntax

#ASM

Notes
All statements between the #ASM and #ENDASM directives are handled by the assembler. This can be used to access uM-FPU64 instructions that aren’t supported directly by the compiler.

Examples

```
#asm
   RTC, INIT+RTCC ; enable RTCC pin
#endasm
```

See Also

#ENDASM
#DEVICE

Defines a loadable device. The device code is loaded from the specified device library file.

Syntax

```
#DEVICE device_file{:device_name}
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>device_file</td>
<td>string</td>
<td>Specifies the name of a Device Library File.</td>
</tr>
<tr>
<td>device_name</td>
<td>string</td>
<td>Specifies the name of the loadable device. If device_name is not specified, then device_name is the same as the device_file name.</td>
</tr>
</tbody>
</table>

The device code is loaded from the specified device library file. A #DEVICE directive and a call to devio(device, LOAD_DEVICE, device_name) must be included in the FPU source file before a loadable device can be used.

Examples

```
#DEVICE sdfat ; loads the SD FAT16/FAT32 device
```
#END

End of user-defined function.

Syntax

#END

Notes

Specifies the end of a user-defined function. If another function is defined immediately after the current function, the #END directive is not required, since the #FUNCTION directive will also end the current function.

Examples

```
#function getID() long ; start of function
    return(23) ; return long integer value = 23
#end ; end of function
```

See Also

#FUNCTION, User-defined Functions
#ENDASM

End of assembler code.

Syntax

```
#ENDASM
```

Notes

All statements between the `#ASM` and `#ENDASM` directives are handled by the assembler. This can be used to access uM-FPU64 instructions that aren’t supported directly by the compiler.

Examples

```
#asm
  RTC, INIT+RTCC ; enable RTCC pin
#endasm
```

See Also

```
#ENDASM
```

#FIRMWARE_REQUIRED

Specifies the minimum uM-FPU64 firmware required for the code.

Syntax

```
#FIRMWARE_REQUIRED, release
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>release</td>
<td>number</td>
<td>Four digit uM-FPU64 release code.</td>
</tr>
</tbody>
</table>

Examples

```
#FIRMWARE_REQUIRED 4050
```
#FUNC

Prototype for user-defined function stored in Flash memory.

Syntax

```
#FUNC number name[(arg1Type, arg2Type, ...)]          user-defined procedure
#FUNC number name([arg1Type, arg2Type, ...]) returnType] user-defined function
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>byte constant %</td>
<td>Assign function to the specified Flash memory function (0-63).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assign function to the next available Flash memory function.</td>
</tr>
<tr>
<td>name</td>
<td>register</td>
<td>Procedure name or function name.</td>
</tr>
<tr>
<td>arg1Type,</td>
<td>register</td>
<td>Argument types. e.g. FLOAT, LONG, ULONG</td>
</tr>
<tr>
<td>arg2Type, ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>returnType</td>
<td>register</td>
<td>Function return type. e.g. FLOAT, LONG, ULONG</td>
</tr>
</tbody>
</table>

Notes

The #FUNC directive is used to define a prototype for user-defined function stored in Flash memory. `Number` specifies where to store the Flash memory function. If a percent character (%) is used in place of `number`, the function will be stored at the next available Flash memory function number. Prototypes should be placed at the start of the program prior to any user-defined functions. The symbol name for the user-defined function (name), the data type of the any arguments (arg1Type, arg2Type, ...), and the data type of the return value (returnType) are defined. The IDE compiler uses this information to generate the code for calls to user-defined functions and procedures.

Examples

See the examples for #FUNCTION.

See Also

#FUNCTION, User-defined Functions
#FUNCTION

Display register value in the debug trace.

Syntax

```c
#FUNCTION number name[(arg1Type, arg2Type, ...) ]
#FUNCTION number name[(arg1Type, arg2Type, ...)] returnType
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>byte constant</td>
<td>Assign function to the specified Flash memory function (0-63).</td>
</tr>
<tr>
<td>name</td>
<td>register</td>
<td>Assign function to the next available Flash memory function.</td>
</tr>
<tr>
<td>arg1Type,</td>
<td>register</td>
<td>Argument types. e.g. FLOAT, LONG, ULONG</td>
</tr>
<tr>
<td>arg2Type,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>returnType</td>
<td>register</td>
<td>Function return type. e.g. FLOAT, LONG, ULONG</td>
</tr>
</tbody>
</table>

Notes

The #FUNCTION directive is used to define user-defined function stored in Flash memory. Number specifies where to store the Flash memory function. If an #FUNC prototype directive was previously defined for this function, number should not be specified. The symbol name for the user-defined function (name), the data type of the any arguments (arg1Type, arg2Type, ...), and the data type of the return value (returnType) are defined. All statements between the #FUNCTION directive and the next #FUNCTION, or #END directive will be compiled and stored as part of the function. If returnType is specified by the directive, the last statement of the function must be a RETURN statement.

Examples

```c
#FUNC 0 getID() long ; Flash memory function at slot 0
#FUNC % getDistance() float ; Flash memory function at next available slot
#FUNC % getLocation(long) ; Flash memory procedure at next available slot

#FUNCTION getID() long ; Flash memory function, returns long
#FUNCTION getDistance() float ; Flash memory function , returns float
#FUNCTION getLocation(long) ; Flash memory procedure
```

See Also

#END, #FUNC, RETURN, User-defined Functions

uM-FPU64 Instruction Set: FCALL, RET, RET,cc
#IDE_REQUIRED

Specifies the minimum IDE release required for the code.

Syntax

```
#IDE_REQUIRED, release
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>release</td>
<td>number</td>
<td>Three digit uM-FPU64 IDE release number.</td>
</tr>
</tbody>
</table>

Examples

```
#IDE_REQUIRED 407
```
#TARGET_CODE

Specify target code link.

**Syntax**

```
#TARGET_CODE link_ID
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>link_ID</td>
<td>string</td>
<td>Specifies a unique link ID.</td>
</tr>
</tbody>
</table>

This directive instructs the compiler to generate a target code link. Target code links allow target files to be automatically updated when the *Update Target File…* button is pressed in the *Output Window*. A begin link is output at the start of linked code and an end link is output at the end of the linked code. The user can define as many links as needed.

**Example**

This source code:

```plaintext
#TARGET_CODE Section1
F1 = F2 + 10
```

Generates the following output for the Arduino target:

```plaintext
// [--- uM-FPU64 ---] Begin Section1
// F1 = F2 + 10
Fpu.write(SELECTA, 1, FSET, 2, FADDI, 10);
//
// [--- uM-FPU64 ---] End Section1
```

If this linked code is copied to the target file, output from future compiles can be automatically copied to the target file using the *Update Target File…* button in the *Output Window*. 
#TARGET_OPTIONS

Specify target options.

Syntax

```markdown
#TARGET_OPTIONS, PICAXE, [X | M2 | X1 | X2], [B0...B55]
#TARGET_OPTIONS, PICMODE
#TARGET_OPTIONS, PROPELLER
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>M2</td>
<td>X1</td>
</tr>
<tr>
<td>B0...B55</td>
<td>register</td>
<td></td>
</tr>
</tbody>
</table>

**PICAXE**

This target option instructs the compiler to generate code that uses the additional registers available on newer PICAXE chips and to determine which register to use for the FPU support routines. The FPU support routines use PICAXE variable B13 by default. If target options are used to change this register, the definitions for the following symbols must also be changed in the support routines.

**PICMODE**

This target option instructs the compiler to generate target code floating point constants in PICMODE format.

**PROPELLER**

This target option instructs the compiler to generate target code using Parallax Propeller syntax.

Examples

```markdown
#TARGET_OPTIONS, PICAXE, X2 ; specifies PICAXE X2
```
#XOP

Defines an XOP (extended opcode) instruction. The XOP definition is loaded from the specified XOP library file.

Syntax

```
#XOP  xop_file{:xop_name}
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xop_file</td>
<td>string</td>
<td>Specifies the name of an XOP Library File.</td>
</tr>
<tr>
<td>xop_name</td>
<td>string</td>
<td>Specifies the name of an XOP instruction. If xop_name is not specified, then xop_name is the same as the xop_file name.</td>
</tr>
</tbody>
</table>

The XOP definition is loaded from the specified XOP library file. An #XOP directive must be included in the FPU source file before an XOP is called. The arguments and return value for the XOP (if any) are defined in the XOP library file. Separate documentation is provided for all the XOP library files which describes each XOP instruction and the arguments and the return value (if any).

Examples

```
#XOP quaternion:q_add  ; loads the q_add XOP instruction
#XOP quaternion:q_norm  ; loads the q_norm XOP instruction
```
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
```

or

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

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.

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax 1</th>
<th>Syntax 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td><code>FADD, reg</code></td>
<td><code>LOG10</code></td>
</tr>
<tr>
<td>SELECTA, reg</td>
<td><code>FSUBR, reg</code></td>
<td><code>EXP</code></td>
</tr>
<tr>
<td>SELECTX, reg</td>
<td><code>FMUL, reg</code></td>
<td><code>EXP10</code></td>
</tr>
<tr>
<td>CLR, reg</td>
<td><code>FDIV, reg</code></td>
<td><code>SIN</code></td>
</tr>
<tr>
<td>CLRA</td>
<td><code>FDIVR, reg</code></td>
<td><code>COS</code></td>
</tr>
<tr>
<td>CLRX</td>
<td><code>FPONI, reg</code></td>
<td><code>TAN</code></td>
</tr>
<tr>
<td>CLR0</td>
<td><code>FCMP, reg</code></td>
<td><code>ASIN</code></td>
</tr>
<tr>
<td>COPY, reg, reg</td>
<td><code>FSET0</code></td>
<td><code>ACOS</code></td>
</tr>
<tr>
<td>COPYA, reg</td>
<td><code>FADD0</code></td>
<td><code>ATAN</code></td>
</tr>
<tr>
<td>COPYX, reg</td>
<td><code>FSUBR0</code></td>
<td><code>ATAN2, reg</code></td>
</tr>
<tr>
<td>LOAD, reg</td>
<td><code>FMUL0</code></td>
<td><code>DEGREES</code></td>
</tr>
<tr>
<td>LOADA</td>
<td><code>FDIV0</code></td>
<td><code>RADIANS</code></td>
</tr>
<tr>
<td>LOADX</td>
<td><code>FDIVR0</code></td>
<td><code>FMOD, reg</code></td>
</tr>
<tr>
<td>ALOADX</td>
<td><code>FPONI0</code></td>
<td><code>FLOOR</code></td>
</tr>
<tr>
<td>XSAVE, reg</td>
<td><code>FSET0</code></td>
<td><code>CEIL</code></td>
</tr>
<tr>
<td>XSAVEA</td>
<td><code>FADD0</code></td>
<td><code>ROUND</code></td>
</tr>
<tr>
<td>COPY0, reg</td>
<td><code>FSUBR0</code></td>
<td><code>FMIN, reg</code></td>
</tr>
<tr>
<td>LCOPYI, bb, reg</td>
<td><code>FADDI, bb</code></td>
<td><code>FMAX, reg</code></td>
</tr>
<tr>
<td>SWAP, reg, reg</td>
<td><code>FSUBI, bb</code></td>
<td><code>FCNV, bb</code></td>
</tr>
<tr>
<td>SWAPA, reg</td>
<td><code>FSUBRI, bb</code></td>
<td><code>FMAC, reg, reg</code></td>
</tr>
<tr>
<td>LEFT</td>
<td><code>FMULI, bb</code></td>
<td><code>FMIN, reg, reg</code></td>
</tr>
<tr>
<td>RIGHT</td>
<td><code>FDIVI, bb</code></td>
<td><code>FMAX, reg, reg</code></td>
</tr>
<tr>
<td>FWRITE, reg, floatval</td>
<td><code>FDIVRI, bb</code></td>
<td><code>LOADBYTE bb</code></td>
</tr>
<tr>
<td>FWRITEA, floatval</td>
<td><code>FPONI, bb</code></td>
<td><code>LOADBYTE bb</code></td>
</tr>
<tr>
<td>FWRITEX, floatval</td>
<td><code>FCMPI, bb</code></td>
<td><code>LOADWORD www</code></td>
</tr>
<tr>
<td>FWRITE0, floatval</td>
<td><code>FSTATUS, reg</code></td>
<td><code>LOADWord www</code></td>
</tr>
<tr>
<td>FREAD</td>
<td><code>FSTATUSA</code></td>
<td><code>LOADE</code></td>
</tr>
<tr>
<td>FREADA</td>
<td><code>FCMPI2, reg, reg</code></td>
<td><code>FCOPYI, bb, reg</code></td>
</tr>
<tr>
<td>FREADX</td>
<td><code>FNEG</code></td>
<td><code>FLOAT</code></td>
</tr>
<tr>
<td>FREAD0</td>
<td><code>FABS</code></td>
<td><code>FIX</code></td>
</tr>
<tr>
<td>ATOF, string</td>
<td><code>FINV</code></td>
<td><code>FIXR</code></td>
</tr>
<tr>
<td>FTOA, bb</td>
<td><code>SQRT</code></td>
<td><code>FRAC</code></td>
</tr>
<tr>
<td>FSET, reg</td>
<td><code>ROOT, reg</code></td>
<td><code>FSPLIT</code></td>
</tr>
<tr>
<td>FADD, reg</td>
<td><code>LOG</code></td>
<td><code>SELECTMA, reg, bb, bb</code></td>
</tr>
</tbody>
</table>
```
SELECTMB, reg, bb, bb
SELECTMC, reg, bb, bb
LOADMA, bb, bb
LOADMB, bb, bb
LOADMC, bb, bb
SAVEA, bb, bb
SAVEB, bb, bb
SAVC, 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
JMP, _label
TABLE, bb
FTABLE, bb
LTABLE, bb
POLY, bb
GOTO, reg
LWRITE, reg, longval
LWRITEA, longval
LWRITEX, longval
LWRITE0, longval
LREAD, reg
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
LSHIFTI, bb
LANDI, bb
LORI, bb
DIGIO, bb
ADCMODE, bb
ADCTRIG
ADCScale, bb
ADCLONG, bb
ADCLOAD, bb
ADCWAIT
LADD0
LSUB0
LMUL0
LDIV0
LCMP0
LUDIV0
RTC, bb
LUCMP0
SETARGS
LTST0
EXTSET
LSETI, bb
EXPLONG
LADDI, bb
LSUBI, bb
LMULI, bb
LDIVI, bb
LCMPI, bb
LUDIVI, bb
STRFIELD, bb
LSTATUS, reg
LSTATUSA
LSTATUS2, reg, reg
LUCMP2, reg, reg
LNEG
LABS
READSEL
STRTF, string
LUCMP2, reg, reg
SYNC
READSTATUS
REGV
LINC, reg
LDEC, reg
LMUL, reg
LDIV, reg
LCMP, reg
LUDIV, reg
RTC, bb
READVAR
SETARGS
LNOT
LOR
LXOR
LSHIFT
LMIN
LMAX
LONGBYTE, bb
LONGUBYTE, bb
LONGWORD, www
LONGUWORD, www
CHECKSUM
BREAK
IEEMODE
PICMODE
VERSION
TRACOFF
TRACEON
TRACESTR, string
TRACEREG, reg
RESET
LWRITE, reg, longval
LWRITEA, longval
LWRITEX, longval
LWRITE0, longval
LREAD, reg
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
LSHIFTI, bb
LANDI, bb
LORI, bb
DIGIO, bb
ADCMODE, bb
ADCTRIG
ADCScale, bb
ADCLONG, bb
ADCLOAD, bb
ADCWAIT
LADD0
LSUB0
LMUL0
LDIV0
LCMP0
LUDIV0
RTC, bb
LUCMP0
SETARGS
LTST0
EXTSET
LSETI, bb
EXPLONG
LADDI, bb
LSUBI, bb
LMULI, bb
LDIVI, bb
LCMPI, bb
LUDIVI, bb
STRFIELD, bb
LSTATUS, reg
LSTATUSA
LSTATUS2, reg, reg
LUCMP2, reg, reg
LNEG
LABS
READSEL
STRTF, string
LUCMP2, reg, reg
SYNC
READSTATUS
REGV
LINC, reg
LDEC, reg
LMUL, reg
LDIV, reg
LCMP, reg
LUDIV, reg
RTC, bb
READVAR
SETARGS
LNOT
LOR
LXOR
LSHIFT
LMIN
LMAX
LONGBYTE, bb
LONGUBYTE, bb
LONGWORD, www
LONGUWORD, www
CHECKSUM
BREAK
IEEMODE
PICMODE
VERSION
TRACOFF
TRACEON
TRACESTR, string
TRACEREG, reg
RESET

Where:
reg
fnum
bb
bb...
dev
wwww
_label
cc
floatval
longval

register number (0-127)
Flash function number (0-63)
8-bit value
multiple 8-bit values
device
16-bit value
address label
condition code (Z,EQ,NZ,NE,LT,LE,GT,GE,PZ,MZ,INF,FIN,PINF,MINF,NAN,TRUE,FALSE)
floating point value
long integer value
string ASCII string

Data Directives
The following data directives can be used to define integer and floating point values. Multiple values can be entered for each data directive.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#BYTE</td>
<td>8-bit data values</td>
</tr>
<tr>
<td>#WORD</td>
<td>6-bit integer values</td>
</tr>
<tr>
<td>#LONG, #LONG32</td>
<td>32-bit integer values</td>
</tr>
<tr>
<td>#LONG64</td>
<td>64-bit integer values</td>
</tr>
<tr>
<td>#FLOAT, #FLOAT32</td>
<td>32-bit floating point values</td>
</tr>
<tr>
<td>#DOUBLE, #FLOAT64</td>
<td>64-bit floating point values</td>
</tr>
</tbody>
</table>

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

#byte, 3, -3, 5, 4
defines three 8-bit integer values

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.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PRINT_FLOAT</td>
<td>print floating point value (if no format specified, 0 is assumed)</td>
</tr>
<tr>
<td>#PRINT_LONG</td>
<td>print integer value (if no format specified, 0 is assumed)</td>
</tr>
<tr>
<td>#PRINT_FPUSTRING</td>
<td>print FPU string</td>
</tr>
<tr>
<td>#PRINT_STRING</td>
<td>print string (e.g. carriage return, linefeed)</td>
</tr>
<tr>
<td>#PRINT_NEWLINE</td>
<td>print new line</td>
</tr>
</tbody>
</table>

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

angle EQU F10
#asm
SELECTA, angle
#endasm

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRA, &lt;label&gt;</td>
<td>branch to label</td>
</tr>
<tr>
<td>BRA, &lt;condition code&gt;, &lt;label&gt;</td>
<td>if condition code is true, branch to label</td>
</tr>
<tr>
<td>JMP, &lt;label&gt;</td>
<td>jump to label</td>
</tr>
<tr>
<td>JMP, &lt;condition code&gt;, &lt;label&gt;</td>
<td>if condition code is true, jump to label</td>
</tr>
<tr>
<td>GOTO, &lt;register&gt;</td>
<td>jump the address contained in the register</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>RET</th>
<th>return from function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET, &lt;condition code&gt;</td>
<td>if condition is true, return from function</td>
</tr>
</tbody>
</table>

### Condition Codes

The condition codes used by various instructions are summarized below.

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

### 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.

```plaintext
_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

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

Assembler Code

```assembly
#asm
SELECTA, tmp
FCMPI, 10
BRA, GE, _1

SELECTA, sum
FADDI, 1
BRA, _2

_1:
SELECTA, sum
FMULI, 10

_2:
#endasm
```

if tmp < 10

sum = sum + 1

else

sum = sum * 10
endif

---

**Repeat Statement**

Psuedocode

repeat 10 times
sum = sum + 1

Assembler Code

```assembly
#asm
SELECTA, cnt
LSETI, 20

_loop:
SELECTA, sum
FADDI, 1

LDEC, cnt
BRA, GT, _loop

#endasm
```

set loop counter to 20

sum = sum + 1

decrement loop counter

repeat until done

---

**For Statement**

Psuedocode

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

Assembler Code

```assembly
#asm
SELECTA, cnt
LSET, startValue

_loop:
SELECTA, sum
FADDI, 1

LINC, cnt
LCMP2, cnt, endValue

#endasm
```

set loop counter to start value

sum = sum + 1

increment loop counter

check for end value
**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.

```asm
STRSET, "test"
```

Special characters can be entered using a backslash prefix. The special characters are as follows:

- `\r` carriage return (0x0D)
- `\n` linefeed (0x0A)
- `\t` horizontal tab (0x09)
- `\v` vertical tab (0x0B)
- `\\` backslash
- `"` double quote
- `\xx` 8-bit value (where xx are hexadecimal digits, e.g. `\0C``

```asm
STRSET, "line1\r\nline2"
```

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

```asm
STRSET, "line1", 13, 10, "line2"
```

An empty string can be specified in two ways.

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

**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
```

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

See the *uM-FPU64 Instruction Set* document for a list of the predefined symbols for matrix operations.
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 \texttt{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\_FPSTRING=<{t}print\_fpuString\{READSTR};>
```
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. 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}
       jsrfpu_write
       lda{byte}+1
       jsrfpu_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_CMD
WRITE_DATA
WRITE_WORD_FORMAT
WRITE_LONG_FORMAT
WRITE_FLOAT_FORMAT
WRITE_STRING_FORMAT
WRITE_BYTE
```
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>
MAX_WRITE=<6>
TAB_SPACING=<-4>
COMMENT_PREFIX=<//>
SOURCE_PREFIX=<{t}// >
HEX_FORMAT=<0x{byte}>
STRING_HEX_FORMAT=<\x{byte}>
```

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:

```c
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 format command instead of a separate function call (i.e. WRITE_WORD_FORMAT instead of WRITE_WORD).

```c
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.

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

The commands for reading data values are shown below.

```c
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.

```c
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.

```c
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.

```c
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:
```c
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.

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

An example of the opcodes generated is as follows:
```c
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.

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

An example of the opcodes generated is as follows:
```c
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_CMD**

Define format for Read Command Instructions

READ_CMD=<string>
Default: empty string
Parameters: {name},{n},{data}
Example: READ_CMD=<{name} := FPU.ReadCmd{n}({data}>

Description: This command defines the format of the read command instructions. Parameter {name} is replaced with the target variable name. Parameter {n} is replaced with the appropriate read command as follows:

ReadCmd, ReadCmdByte, ReadCmdWord, ReadCmdLong, ReadCmd2Long, ReadCmdStr, ReadCmdByte3.

Parameter {data} is replaced with the appropriate data items for the read command.

**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_WORDS

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_TRANSFER=<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_TRANSFER=<string>
Default: empty string
Parameters: none
Example: \texttt{START\_WRITE=\langle\texttt{CALL START\_WRITE();}\rangle}

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

\textbf{STOP\_TRANSFER} \hspace{2cm} Define instructions for end of read or write transfer

\texttt{STOP\_TRANSFER=\langle\texttt{string}\rangle}
Default: empty string
Parameters: none
Example: \texttt{STOP=\langle\texttt{CALL STOP();}\rangle}

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

\textbf{STRING\_HEX\_FORMAT} \hspace{2cm} Define format for non-printable string characters

\texttt{STRING\_HEX\_FORMAT=\langle\texttt{string}\rangle}
Default: empty string
Parameters: none
Example: \texttt{STRING\_HEX\_FORMAT=\langle\texttt{|byte|}\rangle}

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

\textbf{TAB\_SPACING} \hspace{2cm} Set number of characters per tab

\texttt{TAB\_SPACING=\langle n \rangle}
Default: 4
Parameters: none
Example: \texttt{TAB\_SPACING=\langle 8 \rangle}

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

\textbf{TARGET\_NAME} \hspace{2cm} Define the target name

\texttt{TARGET\_NAME=\langle target name \rangle}
Default: none
Parameters: none
Example: \texttt{TARGET\_NAME=\langle C compiler \rangle}
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.

**TARGET_OPTIONS**

Define target options

**TARGET_OPTIONS**=<target, ...>

Default: none

Parameters: device specific

Example: TARGET_OPTIONS= <PICAXE, X2>

Description: The target options are device specific.

**TARGET_OPTIONS**=<PICAXE, {X | M2 | X1 | X2}, {B0...B55}>

{X | M2 | X1 | X2} specifies the type of PICAXE chip used

{B0...B55} specifies the PICAXE variable used by the FPU support routines

This target option instructs the compiler to generate code that uses the additional registers available on newer PICAXE chips and to determine which register to use for the FPU support routines. The FPU support routines use PICAXE variable B13 by default. If target options are used to change this register, the definitions for the following symbols must also be changed in the support routines.

**TARGET_OPTIONS**=<PICMODE>

The default format for floating point constants generated by the compiler is IEEE 754. This target option instructs the compiler to generate target code floating point constants in PICMODE format.

**TARGET_OPTIONS**=<PROPELLER>

This target option instructs the compiler to generate target code using Parallax Propeller syntax.

---

**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_CMD

Define format for Write Command Instructions

WRITE_CMD=<string>
Default: empty string
Parameters: {n},{data}
Example: WRITE_CMD=<FPU.WriteCmd{n}({data})>

Description: This command defines the format of the write command instructions. Parameter {n} is replaced with the appropriate write command suffix as follows:
WriteCmd, WriteCmdByte, WriteCmdByte2, WriteCmdByte3, 
WriteCmdByte4, WriteCmdByteWord, WriteCmdByte2Word, 
WriteCmdByte2Word, WriteCmdByteLong, WriteCmdWord, 
WriteCmdLong, WriteCmdStr, WriteCmdByteStr, WriteCmdByte2Str. 
Parameter \( \{\text{data}\} \) is replaced with the appropriate data items for the write command.

---

**WRITE_DATA**

Define format for Write Data Instructions

**WRITE_DATA=**<string>

Default: empty string

Parameters: \( \{n\}, \{\text{data}\} \)

Example: \( \text{WRITE\_DATA}=<\text{FPU.WriteData}\{n\}\{\text{data}\}> \)

Description: This command defines the format of the write data instructions. Parameter \( \{n\} \) is replaced with the appropriate write data suffix depending on the number of data items.

WriteData1, WriteData2, WriteData3, ...

Parameter \( \{\text{data}\} \) is replaced with the appropriate data items for the write data instructions. The write data instructions have a datatype as the first argument, which specifies the data types of the remaining arguments. This is used by targets that pass all arguments as 32-bit values (e.g. Parallax Propeller). The code generator create the datatype value as a series of is a 2-bit values:

\[
\begin{array}{c|c}
00 & 8\text{-bit data} \\
01 & 16\text{-bit data} \\
10 & 32\text{-bit data} \\
11 & 8\text{-bit opcode}
\end{array}
\]

The datatype bits are specified from left to right, and the value is right justified.

---

**WRITE_LONG**

Define instructions to write 32-bit value

**WRITE_LONG=**<string>

Default: empty string

Parameters: none

Example: \( \text{WRITE\_LONG}=<\text{call fpu.writelong}\{\text{long}\}> \)

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

---

**WRITE_LONG_FORMAT**

Define 32-bit value format for WRITE command

**WRITE_LONG_FORMAT=**<string>

Default: empty string

Parameters: none

Examples:

\[
\begin{align*}
\text{WRITE\_LONG} & = \langle\text{byte}\rangle<<24, \langle\text{byte}\rangle<<16, \langle\text{byte}\rangle<<8, \langle\text{byte}\rangle>
\text{WRITE\_LONG} & = \langle\text{word}\rangle(1), \langle\text{word}\rangle(2)>
\text{WRITE\_LONG} & = \langle\text{long}\rangle
\end{align*}
\]

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_FORMAT=<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_FORMAT=<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.