Introduction
Release 330 of the uM-FPU V3 IDE has a completely rewritten compiler with an expanded set of operators, operator precedence, and control statements. The new code generator produces very efficient code for the FPU. An extensive list of functions and procedures makes it possible to implement most code using high-level code.

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

Operator precedence
~ + -
**
* / %
+ -
<< >>
&
^
|

Control statements
IF...THEN...ELSE
FOR..LOOP
DO...WHILE...UNTIL...LOOP
SELECT...CASE

User-defined functions
• can have up to nine parameters
• can return a value
• can be included in math expressions

Expanded set of functions and procedures
• Math Functions
• ADC Functions
• Serial Input/Output
• String Functions
• Timer Functions
• Matrix Functions
• EEPROM Functions
• External Input / Output
• Miscellaneous Functions
• Debug Functions
Table of Contents

Introduction ........................................................................................................................................ 1
Table of Contents .................................................................................................................................. 2
Summary of Functions .......................................................................................................................... 4
  Control Statements ............................................................................................................................... 4
  Function Directives ............................................................................................................................... 4
  Math Functions ..................................................................................................................................... 5
  ADC Functions ..................................................................................................................................... 5
  Serial Input/Output ............................................................................................................................... 5
  String Functions ................................................................................................................................. 6
  Timer Functions .................................................................................................................................... 6
  Matrix Functions ................................................................................................................................... 6
  EEPROM Functions .............................................................................................................................. 7
  External Input / Output .......................................................................................................................... 7
  Miscellaneous Functions ...................................................................................................................... 7
  Debug Functions .................................................................................................................................. 7
Compiler Reference ............................................................................................................................. 8
  ADCFLOAT .......................................................................................................................................... 8
  ADCLONG .......................................................................................................................................... 8
  ADCMODE .......................................................................................................................................... 9
  ADCSCALE ......................................................................................................................................... 10
  ADCTRIG ........................................................................................................................................... 11
  ADCWAIT ......................................................................................................................................... 11
  BREAK ............................................................................................................................................. 12
  Conditional Expressions ...................................................................................................................... 12
  CONTINUE ........................................................................................................................................ 13
  DO...WHILE...UNTIL...LOOP ............................................................................................................ 13
  EEFLOAT ............................................................................................................................................ 15
  EELONG ............................................................................................................................................ 15
  EESAVE ............................................................................................................................................ 15
  EXIT .................................................................................................................................................. 16
  Expressions ........................................................................................................................................ 16
  EXTLONG .......................................................................................................................................... 18
  EXTSET ............................................................................................................................................ 19
  EXTWAIT ........................................................................................................................................... 19
  FCNV ................................................................................................................................................ 20
  FFT ................................................................................................................................................... 21
  FLOOKUP .......................................................................................................................................... 22
  FOR...NEXT ...................................................................................................................................... 23
  FTABLE ............................................................................................................................................. 24
  FTOA ................................................................................................................................................ 25
  IF...THEN ......................................................................................................................................... 26
  IF...THEN...ELSE ............................................................................................................................... 27
  Line Continuation ............................................................................................................................... 27
  LLOOKUP ......................................................................................................................................... 28
  LOADMA, LOADMB, LOADMC ........................................................................................................... 28
  LTABLE ............................................................................................................................................. 29
  LTOA ................................................................................................................................................ 30
  Math Functions ................................................................................................................................... 31
  MOP .................................................................................................................................................. 32
  POLY ................................................................................................................................................ 40
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>READVAR</td>
<td>41</td>
</tr>
<tr>
<td>RETURN</td>
<td>42</td>
</tr>
<tr>
<td>SAVEMA, SAVEMB, SAVEMC</td>
<td>42</td>
</tr>
<tr>
<td>SELECT...CASE</td>
<td>43</td>
</tr>
<tr>
<td>SELECTMA, SELECTMB, SELECTMC</td>
<td>44</td>
</tr>
<tr>
<td>SERIAL</td>
<td>45</td>
</tr>
<tr>
<td>SETOUT</td>
<td>49</td>
</tr>
<tr>
<td>STATUS</td>
<td>50</td>
</tr>
<tr>
<td>STRBYTE</td>
<td>51</td>
</tr>
<tr>
<td>STRFCHR</td>
<td>51</td>
</tr>
<tr>
<td>STRFIELD</td>
<td>52</td>
</tr>
<tr>
<td>STRFIND</td>
<td>53</td>
</tr>
<tr>
<td>STRFLOAT</td>
<td>53</td>
</tr>
<tr>
<td>STRINC</td>
<td>54</td>
</tr>
<tr>
<td>STRINS</td>
<td>55</td>
</tr>
<tr>
<td>STRLONG</td>
<td>55</td>
</tr>
<tr>
<td>String Constant</td>
<td>56</td>
</tr>
<tr>
<td>STRSEL</td>
<td>56</td>
</tr>
<tr>
<td>STRSET</td>
<td>57</td>
</tr>
<tr>
<td>TICKLONG</td>
<td>57</td>
</tr>
<tr>
<td>TIMELONG</td>
<td>58</td>
</tr>
<tr>
<td>TIMESET</td>
<td>58</td>
</tr>
<tr>
<td>TRACEON, TRACEOFF</td>
<td>59</td>
</tr>
<tr>
<td>TRACEREG</td>
<td>59</td>
</tr>
<tr>
<td>TRACESTR</td>
<td>60</td>
</tr>
<tr>
<td>User-defined Functions</td>
<td>60</td>
</tr>
<tr>
<td>Defining Functions</td>
<td>60</td>
</tr>
<tr>
<td>Passing Parameters and Return Values</td>
<td>61</td>
</tr>
<tr>
<td>Calling Functions</td>
<td>61</td>
</tr>
<tr>
<td>Nested Functions Calls</td>
<td>62</td>
</tr>
<tr>
<td>#EEFUNC</td>
<td>62</td>
</tr>
<tr>
<td>#EEFUNCTION</td>
<td>62</td>
</tr>
<tr>
<td>#END</td>
<td>63</td>
</tr>
<tr>
<td>#FUNC</td>
<td>64</td>
</tr>
<tr>
<td>#FUNCTION</td>
<td>64</td>
</tr>
</tbody>
</table>
Summary of 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]
ENDSELECT

STATUS(conditionCode)

Function Directives

#EEFUNC number name[(arg1Type, arg2Type, ...)]
#EEFUNC number name[(arg1Type, arg2Type, ...)] returnType
#EEFUNCTION [number] name[(arg1Type, arg2Type, ...)]
#EEFUNCTION [number] name[(arg1Type, arg2Type, ...)] returnType
#END

#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 = 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)
result = ROOT(arg1)
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
ADCWAIT

Serial Input/Output

SERIAL(SET_BAUD, baud)
SERIAL(WRITE_TEXT, string)
SERIAL(WRITE_TEXTZ, string)
SERIAL(WRITE_STRBUF)
SERIAL(WRITE_STRSEL)
SERIAL(WRITE_CHAR, value)
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)

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)
Summary of Functions

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 (reg, rows, columns)
SELECTMB (reg, rows, columns)
SELECTMC (reg, rows, columns)

EEPROM Functions
result = EEFLOAT(slot)
result = EEFILEG(slot)
EESAVE(slot, value)

External Input / Output
result = EXTLONG()
EXTSET(value)
EXTWAIT
SETOUT(pin, LOW)
SETOUT(pin, HIGH)
SETOUT(pin, TOGGLE)
SETOUT(pin, HIZ)

Miscellaneous Functions
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)

Debug Functions
BREAK
TRACEON
TRACEOFF
TRACEREG(reg)
TRACESTR(string)
## Compiler Reference

### ADCFLOAT

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

**Syntax**

```plaintext
result = ADCFLOAT(channel)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td><code>float</code></td>
<td>The last ADC reading from the selected channel, multiplied by the scale factor.</td>
</tr>
<tr>
<td>channel</td>
<td><code>long constant</code></td>
<td>ADC channel. (0 or 1)</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**

```plaintext
result = ADCFLOAT(0) ; returns the value for A/D channel 0
; if A/D reading is 200, and scale multiplier = 1.0, result = 200.0
; if A/D reading is 200, and scale multiplier = 1.5, result = 300.0
```

**See Also**

ADCLOAD, ADCMODE, ADCSCALE, ADCTRIG, ADCWAIT

* uM-FPU V3 Instruction Set: ADCLOAD

### ADCLONG

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

**Syntax**

```plaintext
result = ADCLONG(channel)
```

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

**Notes**

Name | result | channel |
---|--------|---------|
Type | `long` | `long constant` |
Description | The last ADC reading from the selected channel. | A/D channel. (0 or 1)
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**

```
result = ADCLONG(0) ; returns the value for A/D channel 0
        ; if A/D channel 0 is 200, result = 200
```

**See Also**

ADCFLOAT, ADCMODE, ADCSCALE, ADCTRIG, ADCWAIT

**uM-FPU V3 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)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat</td>
<td>long constant</td>
<td>The number of additional samples taken at each trigger (0-15).</td>
</tr>
<tr>
<td>period</td>
<td>long expression</td>
<td>The period in microseconds (&gt;= 100).</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCMODE(MANUAL_TRIGGER, 0)</td>
<td>manual trigger, 1 sample per trigger</td>
</tr>
<tr>
<td>ADCMODE(EXTERNAL_TRIGGER, 4)</td>
<td>external input trigger, 5 samples per trigger</td>
</tr>
<tr>
<td>ADCMODE(TIMER_TRIGGER, 0, 1000)</td>
<td>timer trigger every 1000 usec, 1 sample per trigger</td>
</tr>
</tbody>
</table>

See Also

ADCFLOAT, ADCLONG, ADCSCALE, ADCTRIG, ADCWAIT

uM-FPU V3 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 (5V), divided by the the number of ADC steps (the uM-FPU V3.1 FPU has a 12-bit ADC, so there are 4095 steps).

```
ADCSCALE(0, 5/4095) ; set scale factor for channel 0 for range of 0.0 to 5.0
```

See Also

ADCSCALE, ADCMODE, ADCFLOAT, ADCLONG, ADCTRIG, ADCWAIT
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, ADCWAIT

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

```
; 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

uM-FPU V3 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-FPU V3 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**

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

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

**Examples**

```
; 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

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

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

Continues execution at the next iteration of the loop.

*Note:* Must be used inside a `FOR...NEXT` or `DO...WHILE...UNTIL...LOOP` 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`

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
### EEFLOAT

Returns the floating point value from the specified EEPROM slot.

**Syntax**

\[ \text{result} = \text{EEFLOAT}(\text{slot}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>float</td>
<td>The floating point value from the specified EEPROM slot.</td>
</tr>
<tr>
<td>slot</td>
<td>byte constant</td>
<td>EEPROM slot number. (0 to 255)</td>
</tr>
</tbody>
</table>

**Examples**

\[ \text{result} = \text{EEFLOAT}(52) \]; returns the value from EEPROM slot 52

**See Also**

EEFLOAT, EESAVE

uM-FPU V3 Instruction Set: EELOAD

### EELONG

Returns the long integer value from the specified EEPROM slot.

**Syntax**

\[ \text{result} = \text{EELONG}(\text{slot}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>long</td>
<td>The long integer value from the specified EEPROM slot.</td>
</tr>
<tr>
<td>slot</td>
<td>byte constant</td>
<td>EEPROM slot number. (0 to 255)</td>
</tr>
</tbody>
</table>

**Examples**

\[ \text{result} = \text{EELONG}(52) \]; returns the value from EEPROM slot 52

**See Also**

EEFLOAT, EESAVE

uM-FPU V3 Instruction Set: EELOAD

### EESAVE

Store a long integer or floating point value to an EEPROM slot.

**Syntax**

\[ \text{EESAVE}(\text{slot}, \text{value}) \]
Notes
The type of the value expression determines whether a long integer or floating point value is stored.

Examples

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slot</td>
<td>long constant</td>
<td>EEPROM slot number (0 to 255).</td>
</tr>
<tr>
<td>value</td>
<td>long expression</td>
<td>The value to store in the EEPROM slot.</td>
</tr>
</tbody>
</table>

See Also
	EEFLOAT, EELONG
	uM-FPU V3 Instruction Set: EESAVE

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

```
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>`</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Bitwise-XOR</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>Bitwise-AND</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>Shift left</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>Shift right</td>
</tr>
<tr>
<td><code>+</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Modulo operation</td>
</tr>
<tr>
<td><code>**</code></td>
<td>Power</td>
</tr>
<tr>
<td><code>~</code></td>
<td>Ones complement</td>
</tr>
<tr>
<td><code>+</code></td>
<td>Unary plus</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Unary minus</td>
</tr>
</tbody>
</table>

### Operator Precedence

```
~  +  -
**
*  /  %
+  -
<<  >>
&
^   
```

### Syntax

expression:

- bitwise-OR-expression

bitwise-OR-expression:

- bitwise-XOR-expression
  - bitwise-XOR-expression | bitwise-XOR-expression

bitwise-XOR-expression:

- bitwise-AND-expression
  - bitwise-AND-expression ^ bitwise-AND-expression

bitwise-AND-expression:

- shift-expression
  & shift-expression
shift-expression:
    additive-expression
    << | >> additive-expression

additive-expression:
    multiplicative-expression
    + | - multiplicative-expression

multiplicative-expression:
    power-expression
    * | / | % power-expression

power-expression:
    unaryExpression
    ** unaryExpression

unary-expression:
    primary-expression
    - | + | - primary-expression

primary_expression:
    ( expression )
    FLOAT(expression)
    FIX(expression)
    FIXR(expression)
    mathFunction
    userFunction
    register
    variable

Examples

<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>

See Also

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

EXTLONG

Returns the value of the external input counter.

Syntax

result = EXTLONG()

Examples
EXTSET

Sets the value of the external input counter.

Syntax

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

Examples

```
EXTSET(0) ; the external input counter is set to zero
```

See Also

EXTLONG, EXTWAIT

*uM-FPU V3 Instruction Set: EXTLONG*
See Also
EXTLONG, EXTSET
uM-FPU V3 Instruction Set: EXTWAIT

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

Syntax
result = FCNV(value, 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_NM</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 meters²</td>
<td>result = value * 4046.856422</td>
</tr>
<tr>
<td>17</td>
<td>M2_ACRES</td>
<td>meters² to acres</td>
<td>result = value / 4046.856422</td>
</tr>
</tbody>
</table>
Examples

```plaintext
distance = FCNV(200, FT_M) ; returns 60.96 (meters)
tempF = FCNV(100, C_F) ; returns 212.0 (degree fahrenheit)
tempF = FCNV(100, 1) ; returns 212.0 (degree fahrenheit)
```

See Also

*uM-FPU V3 Instruction Set: FCNV*

FFT

Perform a Fast Fourier Transform.

Syntax

```plaintext
FFT(type)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OZ_G</td>
<td>ounces</td>
<td>to grams result = value * 28.34952313</td>
</tr>
<tr>
<td>G_OZ</td>
<td>grams</td>
<td>to ounces result = value / 28.34952313</td>
</tr>
<tr>
<td>LB_KG</td>
<td>pounds</td>
<td>to kilograms result = value * 0.45359237</td>
</tr>
<tr>
<td>KG_LB</td>
<td>kilograms</td>
<td>to pounds result = value / 0.45359237</td>
</tr>
<tr>
<td>USGAL_L</td>
<td>US gallons</td>
<td>to liters result = value * 3.7854111784</td>
</tr>
<tr>
<td>L_USGAL</td>
<td>liters</td>
<td>to US gallons result = value / 3.7854111784</td>
</tr>
<tr>
<td>UKGAL_L</td>
<td>UK gallons</td>
<td>to liters result = value * 0.4546099295</td>
</tr>
<tr>
<td>L_UKGAL</td>
<td>liters</td>
<td>to UK gallons result = value / 0.4546099295</td>
</tr>
<tr>
<td>USOZ_ML</td>
<td>US fluid ounces</td>
<td>to milliliters result = value * 29.57352956</td>
</tr>
<tr>
<td>ML_USOZ</td>
<td>fluid ounces</td>
<td>to US fluid ounces result = value / 3.7854111784</td>
</tr>
<tr>
<td>UKOZ_ML</td>
<td>UK fluid ounces</td>
<td>to milliliters result = value * 28.41312059</td>
</tr>
<tr>
<td>ML_UKOZ</td>
<td>milliliters</td>
<td>to UK fluid ounces result = value / 28.41312059</td>
</tr>
<tr>
<td>CAL_J</td>
<td>calories</td>
<td>to Joules result = value * 4.18605</td>
</tr>
<tr>
<td>J_CAL</td>
<td>Joules</td>
<td>to calories result = value / 4.18605</td>
</tr>
<tr>
<td>HP_W</td>
<td>horsepower</td>
<td>to watts result = value * 745.7</td>
</tr>
<tr>
<td>W_HP</td>
<td>watts</td>
<td>to horsepower result = value / 745.7</td>
</tr>
<tr>
<td>ATM_KP</td>
<td>atmospheres</td>
<td>to kilopascals result = value * 101.325</td>
</tr>
<tr>
<td>KP_ATM</td>
<td>kilopascals</td>
<td>to atmospheres result = value / 101.325</td>
</tr>
<tr>
<td>MMHG_KP</td>
<td>mmHg</td>
<td>to kilopascals result = value * 0.1333223684</td>
</tr>
<tr>
<td>KP_MMHG</td>
<td>kilopascals</td>
<td>to mmHg result = value / 0.1333223684</td>
</tr>
<tr>
<td>DEG_RAD</td>
<td>degrees</td>
<td>to radians result = value * π / 180</td>
</tr>
<tr>
<td>RAD_DEG</td>
<td>radians</td>
<td>to degrees result = value * 180 / π</td>
</tr>
</tbody>
</table>

Micromega Corporation

uM-FPU V3 IDE Release 330
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

    FFT(FIRST_STAGE+REVERSE) ; perform FFT in single instruction

See Also
uM-FPU V3 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

    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.</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. If the index is less than zero, the first item in the table is returned. If the index value is greater than the length of the table, the last item in the table is returned.

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-FPU V3 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**

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

**Name** | **Description**
---|---
register | A register that is incremented or decremented each time through the loop. The register can be a floating point register or a long integer register.

startExpression | A numeric numeric expression for the starting value of register.

endExpression | A numeric numeric expression for the ending value of register.

stepExpression | A numeric numeric expression for the step value of register.

statements | One or more statements to be executed each time through the loop.

**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**

```plaintext
result = FLOOKUP(n, 0, 1.0, 10.0, 100, 1000) ; if n = 2, then 10.0 is returned
```
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, item1, ...</td>
<td>float constant</td>
<td>A list of floating point 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 GE, then the items in the list are compared as follows:

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

Convert floating point value to string.

**Syntax**

```
FTOA(value, format)
```

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

```plaintext
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
muM-FPU V3 Instruction Set: STRINC, STRDEC

---

### IF...THEN

Conditionally executes a statement.

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

#### Syntax

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

#### Name | Description
---|---
condition | Required. A conditional expression.
CONTINUE | Required. The statement is executed if `condition` is true.
EXIT
RETURN
equalsStatement

#### Notes
If the condition is true, the statement is executed.

#### Examples

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

```plaintext
if n then return ; if n is not zero, then return
```
IF...THEN...ELSE

Conditionally executes a statement or group of statements.

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

**Syntax**

```
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**

```
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**

```c
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,</td>
<td>long constant</td>
<td>The list of long integer constants in the table.</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. If the index is less than zero, the first item in the table is returned. If the index value is greater than the length of the table, the last item in the table is returned.

**Examples**

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

**See Also**

FLOOKUP, FTABLE, LTABLE

*uM-FPU V3 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**

```c
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>row</td>
<td></td>
<td></td>
</tr>
<tr>
<td>column</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```
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-FPU V3 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-FPU V3 Instruction Set: LTABLE*

---

**LTOA**

Convert long integer value to string.

**Syntax**

```
LTOA(value, format)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td><em>long expression</em></td>
<td>The long integer value to convert.</td>
</tr>
<tr>
<td>format</td>
<td><em>long constant</em></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 the format byte 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 the format byte is non-zero, a value between 0 and 15 specifies the length of the converted string. The converted string is right justified. If the format byte is positive, leading spaces are used. If the format byte is negative, its absolute value specifies the length of the converted string, and leading zeros are used. If 100 is added to the format value the value is converted as an unsigned long integer, otherwise it is converted as a 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></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**
Math Functions

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

Syntax

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

### Table

<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></td>
<td>Z, NZ, EQ, NE, LT, GE, LE, GT</td>
<td></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

<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>angle (arg1) converted from radians to degrees.</td>
</tr>
</tbody>
</table>
RADIANS(arg1) Float Float angle arg1 converted from degrees to radians.
FLOOR(arg1) Float Float floor of arg1.
CEIL(arg1) Float Float ceiling of arg1.
ROUND(arg1) Float Float arg1 rounded to the nearest integer.
POWER(arg1, arg2) Float Float arg1 raised to the power of arg2.
ROOT(arg1, arg2) Float Float arg2 root of arg1.
FRAC(arg1) Float Float fractional part of arg1.
INV(arg1) Float Float the inverse of arg1.
FLOAT(arg1) Long Float converts arg1 from long to float.
FIX(arg1) Float Long converts arg1 from float to long.
FIXR(arg1) Float Long rounds arg1 then converts from float to long.
ABS(arg1) Float/Long Float/Long absolute value of arg1.
MOD(arg1, arg2) Float/Long Float/Long the remainder of arg1 divided by arg2.
MIN(arg1, arg2) Float/Long Float/Long the minimum of arg1 and arg2.
MAX(arg1, arg2) Float/Long Float/Long the maximum of arg1 and arg2.

Examples

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

See Also

**uM-FPU V3 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.

```
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(COPYBC)
MOP(COPYBA)
MOP(COPYCA)
MOP(COPYCB)
return = MOP(DETERMINANT)
MOP(LOADRA)
MOP(LOADRB)
MOP(LOADRC)
MOP(LOADBA)
MOP(LOADCA)
MOP(SAVEAR)
MOP(SAVEAB)
MOP(SAVEAC)

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

uM-FPU V3 Instruction Set: MOP

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.

\[ MA[\text{row, column}] = MA[\text{row, column}] / MB[\text{row, column}] \]

**EWISE_DIVR**

Divide each element of matrix B by the element of matrix A.

\[ MA[\text{row, column}] = MB[\text{row, column}] / MA[\text{row, column}] \]

**EWISE.Pow**

Each element of matrix A is raised to the power of the element of matrix B.

\[ MA[\text{row, column}] = MA[\text{row, column}] ** MB[\text{row, column}] \]

### Examples

**Syntax**

```
MOP(EWISE_DIV)
```

; each elements of matrix A is divided by the element in matrix B

### 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 rows in matrix B is the same as the number of columns in matrix C. The size of matrix MA will be updated to reflect the rows and columns of the resulting matrix.

**Syntax**

```
MOP(MULTIPLY)
```

**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.

**Syntax**

```
MOP(IDENTITY)
```

**Syntax**

```
MOP(DIAGONAL, value)
```

; sets matrix A to the identity matrix
<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**

```
MOP(DIAGONAL, 100.0) ; set matrix A to a diagonal matrix with 100.0 stored on the diagonal
```

**Syntax**

```
MOP(TRANSPOSE)
```

**Notes**

Sets matrix A to the transpose of matrix B.

**Examples**

```
MOP(TRANSPOSE) ; 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>long</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 the count, sum, average, minimum, or maximum of all elements in matrix A.

**Examples**
Syntax

MOP(COPYAB)
MOP(COPYAC)
MOP(COPYBA)
MOP(COPYBC)
MOP(COPYCA)
MOP(COPYCB)

Notes

Copies one matrix to another.

Examples

MOP(COPYAB) ; copies matrix A to matrix B

Syntax

return = MOP(DETERMINANT)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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

value = MOP(DETERMINANT) ; return the determinant of matrix A

Syntax

MOP(INVERSE)

Notes

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

Examples

MOP(INVERSE) ; sets matrix A to the inverse of matrix B
### Syntax

- \texttt{MOP(LOADRA, idx1, idx2, ...)}
- \texttt{MOP(LOADRB, idx1, idx2, ...)}
- \texttt{MOP(LOADRC, idx1, idx2, ...)}

### 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:

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

Assuming register 1 contains the value \(\sin A\), and register 2 contains the value \(\cos A\), the following instructions create the matrix.

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

### Syntax

- \texttt{MOP(LOADBA, idx1, idx2, ...)}
- \texttt{MOP(LOADCA, idx1, idx2, ...)}

### 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.
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:

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>g</td>
<td>h</td>
<td>i</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>
```

Examples

```
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(LOADBA, 0, 1, 3, 4); copies the subset shown above from matrix A to matrix B
```

**Syntax**

```
MOP(SAVEAR, 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:

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>g</td>
<td>h</td>
<td>i</td>
</tr>
</tbody>
</table>
```

Examples

```
MOP(SAVEAR, 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

\texttt{MOP(SAVEAB, idx1, idx2, ...)}
\texttt{MOP(SAVEAC, 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.

POLY

Calculates the \(n\)th order polynomial of the floating point value.  

\textit{Note:} Must be used inside a user-defined procedure or function.

Syntax

\texttt{result = POLY(value, coeff1, 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 \texttt{POLY} function can only be used inside an FPU function. The general form of the polynomial is:  
\[
A_n x^n + A_{n-1} x^{n-1} + \ldots + A_0 x^0
\]

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

\begin{verbatim}
value = POLY(x, 3.0, 5.0) ; value = 3x + 5
value = POLY(x, 1, 0, 0, 1) ; value = x^3 + 1
\end{verbatim}

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:

\[
RH_{\text{linear}} = -4.0 + 0.0405 \cdot SO_{\text{RH}} + (-2.8 \cdot 10^6 \cdot SO_{\text{RH}}^2)
\]
The following example makes this calculation.

\[
\text{RHlinear} = \text{POLY}(\text{SORh}, -2.8E-6, 0.0405, -4)
\]

See Also

*uM-FPU V3 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**

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

**Examples**

\[
\text{value} = \text{READVAR}(15) ; \text{returns the start of the string selection point}
\]

See Also
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><em>long expression</em></td>
<td>The value returned from a user-defined function.</td>
</tr>
<tr>
<td></td>
<td><em>float expression</em></td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

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

**Examples**

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

**See Also**

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

SAVEMA, SAVEMB, SAEMC

Store a matrix value.

**Syntax**

```
SAVEMA(row, column, value)
SAVEMB(row, column, value)
SAEMC(row, column, value)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>row</td>
<td><em>long constant</em></td>
<td>The row number of the matrix.</td>
</tr>
<tr>
<td>column</td>
<td><em>long constant</em></td>
<td>The column number of the matrix.</td>
</tr>
<tr>
<td>value</td>
<td><em>float expression</em></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**


### 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
  statements
[CASE compareValue [, compareValue]...
  statements]...
[ELSE statements]
ENDSELECT
```

#### Name

<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 <code>compareValue</code> is equal to the value of <code>compareItem</code>.</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 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

```
SELECTMA(100, 3,3) ; matrix A is defined as a 3x3 matrix starting at register 100
MOP(SCALAR_SET, 0) ; set all values in matrix A to zero
SAVEMA(0, 2, pi) ; store the value pi at row 0, column 2
```
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

SELECTMA, SELECTMB, SELECTMC
Select the registers used for matrix operations.

Syntax
SELECTMA(reg, rows, columns)
SELECTMB(reg, rows, columns)
SELECTMC(reg, rows, columns)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>register</td>
<td>The first register of the matrix.</td>
</tr>
<tr>
<td>rows</td>
<td>long constant</td>
<td>The number of rows.</td>
</tr>
<tr>
<td>columns</td>
<td>long constant</td>
<td>The number of columns.</td>
</tr>
</tbody>
</table>

Notes
The `reg` parameter is the first register of the array. 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTMA(100, 3,3)</td>
<td>matrix A is defined as a 3x3 matrix starting at register 100</td>
</tr>
<tr>
<td>SELECTMB(109, 2,3)</td>
<td>matrix B is defined as a 2x3 matrix starting at register 109</td>
</tr>
<tr>
<td>SELECTMC(115, 3,1)</td>
<td>matrix C is defined as a 3x1 matrix starting at register 115</td>
</tr>
</tbody>
</table>

### See Also

- MOP, LOADMA, LOADMB, LOADMC, SAVEMA, SAVEMB, SAVEMC
- uM-FPU V3 Instruction Set: SELECTMA, SELECTMB, SELECTMC

#### 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_TEXT, string)
SERIAL(WRITE_TEXTZ, string)
SERIAL(WRITE_STRBUF)
SERIAL(WRITE_STRSEL)
SERIAL(WRITE_CHAR, value)
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-FPU V3 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>
</table>
| baud  | `long constant` | The baud rate for the SEROUT and SERIN pins.  
0, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, or 115200 |

### 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_TEXT, string)```

<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 SEROUT pin.</td>
</tr>
</tbody>
</table>

Notes

Writes the string to the SEROUT pin.

Examples

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

Syntax

```SERIAL(WRITE_TEXTZ, string)```

<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 SEROUT pin.</td>
</tr>
</tbody>
</table>

Notes

Writes the string to the SEROUT pin, followed by a zero byte.

Examples

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

Syntax

```SERIAL(WRITE_STRBUF)```

Notes

Writes the contents of the string buffer to the SEROUT pin.

Examples

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

SERIAL(WRITE_STRSEL)

Notes
Writes the current string selection to the SEROUT pin.

Examples

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

Syntax

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 of the value is output to the SEROUT pin.</td>
</tr>
</tbody>
</table>

Notes
Writes the lower 8 bits of the value to the SEROUT pin.

Examples

SERIAL(WRITE_CHAR, $32) ; sends $32 (the digit 2) to the SEROUT pin
SERIAL(WRITE_CHAR, value) ; sends the lower 8 bits of value to the SEROUT pin

Syntax

SERIAL(DISABLE_INPUT)

Notes
The SERIN pin is disabled.

Examples

SERIAL(DISABLE_INPUT) ; disables the SERIN pin

Syntax

SERIAL(ENABLE_CHAR)

Notes
The SERIN 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
**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**

```plaintext
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.

A known problem with the V3.1 chip is that it carriage return characters (0x0D) are returned as zero bytes.

**Examples**

```plaintext
ch = SERIAL(READ_CHAR) ; returns the next serial input character
```

---

**Syntax**

```
SERIAL(ENABLE_NMEA)
```

**Notes**

The SERIN 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

```c
SERIAL(ENABLE_NMEA) ; enables the SERIN 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

```c
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

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

**SETOUT**

Set the OUT0 or OUT1 output pin.

Syntax

```
SETOUT(pin, LOW)
SETOUT(pin, HIGH)
SETOUT(pin, TOGGLE)
```
### SETOUT(pin, HIZ)

<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 (0 or 1).</td>
</tr>
<tr>
<td>action</td>
<td>long constant</td>
<td></td>
</tr>
</tbody>
</table>

#### Notes
The output pins OUT0 and OUT1 are set according to the action specified.

- **LOW**: set output pin low
- **HIGH**: set output pin low
- **TOGGLE**: toggle the output pin
- **HIZ**: set output pin to high impedance

#### Examples

```
SETOUT(0, LOW) ; set OUT0 to low
SETOUT(1, TOGGLE) ; toggle the value of OUT0
SETOUT(0, HIZ) ; set OUT0 to low
```

### 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 the specified condition code. If the NMEA_CHECKSUM or NMEA_OVERRUN condition code is specified, the STATUS condition is true if the corresponding bit is set.

The condition code symbols are as follows:
- Z, NZ, EQ, NE, LT, GE, LE, GT, INF, FIN, PLUS, MINUS, NAN, NOTNAN
- NMEA_CHECKSUM, NMEA_OVERRUN

#### Examples
```
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 expresion</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 the {} 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-FPU V3 Instruction Set: STRBYTE

---

**STRFCHR**

Sets the field separator characters used by the STRFIELD procedure.

**Syntax**

```
STRFCHR(string)
```
### 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 string is not important.

### Examples

See the examples for STRFIELD.

### See Also

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

*uM-FPU V3 Instruction Set: STRINC, STRDEC*

---

**STRFIELD**

Find the specified field in the string.

#### Syntax

```
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.

#### 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 separators
STRFIELD(1)                        ; string buffer = {2010}-7-20 10:57
year = STRLONG()                   ; convert string to year
STRFIELD(5)                        ; string buffer = 2010-7-20 10:{57} pm
minutes = STRLONG()                ; convert string to minutes
```
See Also

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

_uM-FPU V3 Instruction Set: STRINC, STRDEC_

---

**STRFIND**

Find the string in the current string selection.

**Syntax**

```
STRFIND(string)
```

**Table**

<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 string. If the 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.

```
STRSET("abcdef")                        ; string buffer = abcdef{}
STRSEL(0,127)                           ; string buffer = {abcdef}
STRFIND("d")                            ; string buffer = abc{d}ef
```

---

See Also

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

_uM-FPU V3 Instruction Set: STRINC, STRDEC_

---

**STRFLOAT**

Returns the floating point value of the current string selection.

**Syntax**

```
result = STRFLOAT()
```

**Table**

<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 value. 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.

---
STRINC

Increment or decrement the string selection point.

Syntax

```
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.

```
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{}bcd{ef
n = 3
STRINC(n)         ; string buffer = axbcd{ef
STRINS("y")      ; string buffer = axbcdy{ef
```

See Also

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

_uM-FPU V3 Instruction Set: STRTOF_

---

See Also

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

_uM-FPU V3 Instruction Set: STRINC, STRDEC_
STRINS

Insert string at the string selection point.

Syntax

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

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{STRSET}(&quot;abcd&quot;)</td>
<td>string buffer = abcd{}</td>
</tr>
<tr>
<td>\text{STRSEL}(1, 2)</td>
<td>string selection = a{bc}d</td>
</tr>
<tr>
<td>\text{STRINS}(&quot;x&quot;)</td>
<td>string buffer = ax{}d</td>
</tr>
<tr>
<td>\text{STRINS}(&quot;yz&quot;)</td>
<td>string buffer = axy{}zd</td>
</tr>
</tbody>
</table>

See Also

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

uM-FPU V3 Instruction Set: STRINC

STRLONG

Returns the long integer value of the current string selection.

Syntax

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

<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 value. Conversion stops at the first character that is not a valid character for a long integer number.

Examples

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
See Also

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

uM-FPU V3 Instruction Set: STRTOL

String Constant

String constants are used as arguments to some of the string procedures and for string comparisons. A string constant is enclosed in double quote characters. Special characters can be entered using a backslash followed by two hexadecimal digits. The backslash and double quote characters can be entered by preceding them with a backslash.

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 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.

Examples

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

Copy the string to the string buffer.

Syntax

```
STRSET(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 store in string buffer.</td>
</tr>
</tbody>
</table>

Notes

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

Examples

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

```
STRSET("abcd") ; string buffer = abcd{}
```

See Also

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

uM-FPU V3 Instruction Set: STRING, STRDEC

TICKLONG

Returns the number of milliseconds that have elapsed since the FPU timer was started.

Syntax

```
result = 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 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 counter is a 32-bit register.

**Examples**

```
result = TICKLONG() ; returns the number of msec since the FPU timer was started
```

**See Also**

`TIMELONG, TIMESET`

*uM-FPU V3 Instruction Set: TICKLONG*

---

**TIMELONG**

Returns the number of seconds that have elapsed since the FPU timer was started.

**Syntax**

```
result = TIMELONG()
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td><code>long</code></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 second counter is a 32-bit register.

**Examples**

```
result = TIMELONG() ; returns the number of seconds since the FPU timer was started
```

**See Also**

`TICKLONG, TIMESET`

*uM-FPU V3 Instruction Set: TIMELONG*

---

**TIMESET**

Set internal timer values.

**Syntax**

```
TIMESET(seconds)
```

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

**Notes**

The internal seconds timer is set to the value specified and the internal millisecond timer is set to zero.
Examples

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

See Also

TICKLONG, TIMELONG

*uM-FPU V3 Instruction Set: TIMESET*

---

TRACEON, TRACEOFF

Turn the debug instruction trace on or off.

Syntax

```
TRACEON
TRACEOFF
```

Notes

These procedure 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

```
TIMESET(0) ; set seconds timer and msec timer to zero

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-FPU V3 Instruction Set: TRACEON, TRACEOFF*

---

TRACEREG

Display register value in the debug trace.

Syntax

```
TRACEREG(reg)
```

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

mnt = 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-FPU V3 Instruction Set: TRACESTR*

---

**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 message 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-FPU V3 Instruction Set: TRACESTR*

---

**User-defined Functions**

User-defined functions can be stored in Flash memory and EEPROM memory on the uM-FPU V3.1 chip.

**Defining Functions**
The #FUNCTION and #EEFUNCTION directives are used to define Flash memory and EEPROM functions. All statements between the #FUNCTION and #EEFUNCTION directives and the next #FUNCTION, #EEFUNCTION, or #END directive will be compiled and stored as part of the function.

The #FUNC and #EEFUNC directives 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.

### 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, #FUNCTION, #EEFUNC, or #EEFUNCTION 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.
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 \texttt{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
- \#EEFUNC, \#EEFUNCTION, \#END, \#FUNC, \#FUNCTION
- uM-FPU V3 Instruction Set: EECALL, FCALL, RET, RET,cc

\#EEFUNC
Prototype for user-defined function stored in EEPROM.

Syntax
\begin{verbatim}
#EEFUNC number name[(arg1Type, arg2Type, ...)] user-defined procedure
#EEFUNC number name[(arg1Type, arg2Type, ...)] returnType user-defined function
\end{verbatim}

<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 EEPROM slot (0-255).</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>Assign function to the next available EEPROM slot.</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. \texttt{FLOAT, LONG, ULONG}</td>
</tr>
<tr>
<td>arg2Type,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>returnType</td>
<td>Function return type. e.g. \texttt{FLOAT, LONG, ULONG}</td>
</tr>
</tbody>
</table>

Notes
The \#EEFUNC directive is used to define a prototype for user-defined function stored in EEPROM. The number specifies the EEPROM slot that the function is stored in. If a percent character (%) is used in place of the number, the function will be stored at the next available EEPROM slot. Prototypes should be placed at the start of the program prior to any user-defined functions. \texttt{Number} specifies the EEPROM slot that the function is stored in. The symbol name for the user-defined function (\texttt{name}), the data type of the any arguments (\texttt{arg1Type, arg2Type, ...}), and the data type of the return value (\texttt{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 \#EEFUNCTION.

See Also
- \#EEFUNCTION, \#FUNC, \#FUNCTION, \textit{User-defined Functions}

\#EEFUNCTION
Function definition for user-defined EEPROM function.

Syntax
\begin{verbatim}
#EEFUNCTION [number] name[(arg1Type, arg2Type, ...)] user-defined procedure
#EEFUNCTION [number] name[(arg1Type, arg2Type, ...)] returnType user-defined function
\end{verbatim}
## Notes

The `#EEFUNCTION` directive is used to define user-defined function stored in EEPROM. *Number* specifies the EEPROM slot that the function is stored in. If an `#EEFUNC` prototype directive was previously defined for this function, *number* should not be specified. The number specifies the EEPROM slot that the function is stored in. 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 `#EEFUNCTION` directive and the next `#FUNCTION`, `#EEFUNCTION`, 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

<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 EEPROM slot (0-255).</td>
</tr>
<tr>
<td>name</td>
<td>register</td>
<td>Procedure name or function name.</td>
</tr>
<tr>
<td>arg1Type, arg2Type, ...</td>
<td>register</td>
<td>Argument types. e.g. FLOAT, LONG, ULONG</td>
</tr>
<tr>
<td>returnType</td>
<td>register</td>
<td>Function return type. e.g. FLOAT, LONG, ULONG</td>
</tr>
</tbody>
</table>

### See Also

`#EEFUNC, #END, #FUNC, #FUNCTION, RETURN, User-defined Functions`

*uM-FPU V3 Instruction Set: EECALL, FCALL, RET, RET,cc*
**#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). Assign function to the next available EEPROM slot.</td>
</tr>
<tr>
<td>name</td>
<td>register</td>
<td>Procedure name or function name.</td>
</tr>
<tr>
<td>arg1Type, arg2Type, ...</td>
<td>register</td>
<td>Argument types. e.g. FLOAT, LONG, ULONG</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**

#EEFUNC, #EEFUNCTION, #FUNCTION, User-defined Functions

---

**#FUNCTION**

Display register value in the debug trace.

**Syntax**

```
#FUNCTION number name[(arg1Type, arg2Type, ...)]  user-defined procedure
#FUNCTION 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>reg</td>
<td>register</td>
<td>Register to trace.</td>
</tr>
</tbody>
</table>

---

See Also

#EEFUNCTION, #FUNCTION, User-defined Functions
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, #EEFUNCTION, 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

```
#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

#EEFUNC, #EEFUNCTION, #END, #FUNC, RETURN, *User-defined Functions*

uM-FPU V3 Instruction Set: EECALL, FCALL, RET, RET,cc