Micromega Corporation

uM-FPU V3 IDE

Integrated Development Environment

Compiler

Introduction

A completely rewritten compiler was added in release 330 of the uM-FPU V3 IDE. The new compiler has 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

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

ADC Functions

```
result = ADCFLOAT(channel)
result = ADCLONG(channel)
ADCMODE(MANUAL_TRIGGER, repeat)
ADCMODE(EXTERNAL_TRIGGER, repeat)
ADCMODE(TIMER_TRIGGER, repeat, period)
ADCMODE(DISABLE)
ADCSCALE(channel, scaleFactor)
ADCTRIG
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()
STRINC(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)
```

```
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 = EELONG(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

`result = **ADCFLOAT**(channel)`

Name	Type	Description
`result`	*float*	The last ADC reading from the selected channel, multiplied by the scale factor.
`channel`	*long constant*	ADC channel. (0 or 1)

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

```
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

`ADCLONG, 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

`result = **ADCLONG**(channel)`

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

Notes

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

Examples

```
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)
```

Name	Type	Description
`repeat`	*long constant*	The number of additional samples taken at each trigger (0-15).
`period`	*long expression*	The period in microseconds (>= 100).

Notes

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

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

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

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

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

Examples

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

See Also

`ADCFLOAT, ADCLONG, ADCSCALE, ADCTRIG, ADCWAIT`
uM-FPU V3 Instruction Set: `ADCMODE`

ADCSCALE

Sets the scale value for the ADC channel.

Syntax

`ADCSCALE``(channel, scaleFactor)`

Name	Type	Description
`channel`	*long constant*	ADC channel (0 or 1).
`scaleFactor`	*float expression*	Scale factor.

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

`ADCFLOAT, ADCLONG, ADCMODE, ADCTRIG, ADCWAIT`
uM-FPU V3 Instruction Set: `ADCSCALE`

ADCTRIG

Triggers an ADC conversion.

Syntax

`ADCTRIG`

Notes

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

Examples

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

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

See Also

`ADCFLOAT, ADCLONG, ADCMODE, ADCSCALE, ADCTRIG, ADCWAIT`
uM-FPU V3 Instruction Set: `ADCTRIG`

ADCWAIT

Waits until the next ADC value is ready.

Syntax

`ADCWAIT`

Notes

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

Examples

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

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

See Also

`ADCFLOAT, ADCLONG, ADCMODE, ADCSCALE, ADCTRIG`
uM-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

```
x  equ  F10
n  equ  L11

if log(x) < 0.3 then n = n + 1

if n then exit

if n > 1 AND n < 5 then x = 0

if NOT (n > 1 AND n < 10) or n = 5 then continue

if strfield(1) = "GPRMC" then
  ; statements
endif

if status(GT) then return
```

See Also

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

CONTINUE

Continues execution at the next iteration of the loop.
Note: Must be used inside a `FOR...NEXT` or `DO...WHILE...LOOP...UNTIL` control statement.

Syntax

`CONTINUE`

Notes

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

Examples

```
n  equ  L10
x  equ  F11

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

See Also

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

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
```

Name	Description
condition1	While this condition is true, execute the statements in the loop.
statements	One or more statements to be executed each time through the loop.
condition2	While this condition is false, repeat the loop.

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                                              ; infinite loop
  ; statements executed each loop iteration
LOOP
```

```
WHILE n > 0                                     ; loop while n > 0
  ; statements executed each loop iteration
LOOP
```

```
DO                                              ; loop until n > 0
  ; statements executed each loop iteration
UNTIL n > 0
```

```
DO WHILE n >= 10                                ; loop while n >= 10 and n <= 20
  ; 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

`result = **EEFLOAT**(slot)`

Name	Type	Description
`result`	*float*	The floating point value from the specified EEPROM slot.
`slot`	*byte constant*	EEPROM slot number. (0 to 255)

Examples

```
result = EEFLOAT(52)                 ; returns the value from EEPROM slot 52
```

See Also

`EELONG, EESAVE`
uM-FPU V3 Instruction Set: `EELOAD`

EELONG

Returns the long integer value from the specified EEPROM slot.

Syntax

`result = **EELONG**(slot)`

Name	Type	Description
`result`	*long*	The long integer value from the specified EEPROM slot.
`slot`	*byte constant*	EEPROM slot number. (0 to 255)

Examples

```
result = 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

`**EESAVE**(slot, value)`

Name	Type	Description

`slot`	*long constant*	EEPROM slot number (0 to 255).
`value`	*long expression* *float expression*	The value to store in the EEPROM slot.

Notes

The type of the value expression determines whether a long integer or floating point value is stored.

Examples

```
EESAVE(1,1.5)          ; store the floating point value 1.5 to EEPROM slot 1
EESAVE(2,100)          ; store the long integer value 100 to EEPROM slot 2
```

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:

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

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

Syntax

expression:
    bitwise-OR-expression

bitwise-OR-expression:
    bitwise-XOR-expression
    | *bitwise-XOR-expression*

bitwise-XOR-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

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

See Also

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

EXTLONG

Returns the value of the external input counter.

Syntax

`result = **EXTLONG**()`

Name	Type	Description
`result`	*long*	The value of the external input counter.

Examples

```
result = EXTLONG()          ; returns the value from the external input counter
```

See Also

EXTSET, EXTWAIT
uM-FPU V3 Instruction Set: EXTLONG

EXTSET

Sets the value of the external input counter.

Syntax

EXTSET(value)

Name	Type	Description
value	*long expression*	The external input counter is set to this value.

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: EXTSET

EXTWAIT

Wait for the next external input to occur.

Syntax

EXTWAIT

Notes

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

Examples

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

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

See Also

EXTLONG, EXTSET
uM-FPU V3 Instruction Set: EXTWAIT

FCNV

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

Syntax

```
result = FCNV(value, conversion)
```

Name	Type	Description
result	*float*	The converted value.
value	*float expression*	The value to convert.
conversion	*long constant*	The conversion number or conversion symbol. (see list below)

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.

Conversion Number	Conversion Symbol	Description	Conversion
0	F_C	Fahrenheit to Celsius	result = value * 1.8 + 32
1	C_F	Celsius to Fahrenheit	result = (value - 32) * 1.8
2	IN_MM	inches to millimeters	result = value * 25.4
3	MM_IN	millimeters to inches	result = value / 25.4
4	IN_CM	inches to centimeters	result = value * 2.54
5	CM_IN	centimeters to inches	result = value / 2.54
6	IN_M	inches to meters	result = value * 0.0254
7	M_IN	meters to inches	result = value / 0.0254
8	FT_M	feet to meters	result = value * 0.3048
9	M_FT	meters to feet	result = value / 0.3048
10	YD_M	yards to meters	result = value * 0.9144
11	M_YD	meters to yards	result = value / 0.9144
12	MILES_KM	miles to kilometers	result = value * 1.609344
13	KM_MILES	kilometers to miles	result = value / 1.609344
14	NM_M	nautical miles to meters	result = value * 1852.0
15	M_NM	meters to nautical miles	result = value / 1852.0
16	ACRES_M2	acres to meters^2	result = value * 4046.856422
17	M2_ACRES	meters^2 to acres	result = value / 4046.856422
18	OZ_G	ounces to grams	result = value * 28.34952313
19	G_OZ	grams to ounces	result = value / 28.34952313
20	LB_KG	pounds to kilograms	result = value * 0.45359237
21	KG_LB	kilograms to pounds	result = value / 0.45359237

22	USGAL_L	US gallons to liters	result = value * 3.7854111784
23	L_USGAL	liters to US gallons	result = value / 3.7854111784
24	UKGAL_L	UK gallons to liters	result = value * 4.546099295
25	L_UKGAL	liters to UK gallons	result = value / 4.546099295
26	USOZ_ML	US fluid ounces to milliliters	result = value * 29.57352956
27	ML_USOZ	milliliters to US fluid ounces	result = value / 29.57352956
28	UKOZ_ML	UK fluid ounces to milliliters	result = value * 28.41312059
29	ML_UKOZ	milliliters to UK fluid ounces	result = value / 28.41312059
30	CAL_J	calories to Joules	result = value * 4.18605
31	J_CAL	Joules to calories	result = value / 4.18605
32	HP_W	horsepower to watts	result = value * 745.7
33	W_HP	watts to horsepower	result = value / 745.7
34	ATM_KP	atmospheres to kilopascals	result = value * 101.325
35	KP_ATM	kilopascals to atmospheres	result = value / 101.325
36	MMHG_KP	mmHg to kilopascals	result = value * 0.1333223684
37	KP_MMHG	kilopascals to mmHg	result = value / 0.1333223684
38	DEG_RAD	degrees to radians	result = value * π / 180
39	RAD_DEG	radians to degrees	result = value * 180 / π

Examples

```
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

FFT(type)

Name	Type	Description
type	*long constant*	The type of FFT operation: FIRST_STAGE NEXT_STAGE NEXT_LEVEL NEXT_BLOCK Modifiers: +REVERSE bit reverse sort pre-processing +PRE pre-processing for inverse FFT +POST post-processing for inverse FFT

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, ...)`

Name	Type	Description
`result`	*float*	The returned value.
`value`	*long expression*	The lookup index for the lookup table.
`item0,` `item1, ...`	*float constant*	The list of floating point constants for the lookup 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 = 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

```
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

```
n  equ  L10
x  equ  F11

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

```
n   equ  L10
x   equ  F11

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

See Also

`CONTINUE, DO...WHILE...UNTIL...LOOP, EXIT, IF...THEN, IF...THEN...ELSE, RETURN, SELECT...CASE`

FTABLE

Returns the index of the first item in the list that satisfies the condition code.
Note: Must be used inside a user-defined procedure or function.

Syntax

`result = **FTABLE**(value, cc, item0, item1, ...)`

Name	Type	Description
`result`	*long*	The index of the first item in the lookup table that satisfies the condition.
`value`	*float expression*	The floating point value to compare with the table items.
`cc`	*condition code*	Condition code. `Z, NZ, EQ, NE, LT, GE, LE, GT`
`item0, item1, ...`	*float constant*	A list of floating point constants for the lookup 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:

value >= item0
value >= item1
value >= item2
...

```
index = FLOOKUP(value, GE, 1.0, 5.5, 10.0, 100.0)      ; if value = 1, index = 0
                                                       ; if value = 17.5, index = 2
```

See Also

`FLOOKUP, LLOOKUP, LTABLE`
uM-FPU V3 Instruction Set: `FTABLE`

FTOA

Convert floating point value to string.

Syntax

FTOA(value, format)

Name	Type	Description
value	*float expression*	The floating point value to convert.
format	*long constant*	The format specifier.

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:

```
1.0              NaN              0.0
10e20            Infinity         -0.0
3.1415927        -Infinity        1.0
-52.333334       -3.5e-5          0.01
```

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)

Value in register A	*Format byte*	*Display format*
123.567	61 (6.1)	`  123.6`
123.567	62 (6.2)	`123.57`
123.567	42 (4.2)	`*.**`
0.9999	20 (2.0)	` 1`
0.9999	31 (3.1)	`1.0`

Examples

In the following example the [] characters are used to shown the string selection point.

```
x    equ  F10

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

See Also

LTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSEL, STRSET
uM-FPU V3 Instruction Set: STRINC, STRDEC

IF...THEN

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

Syntax

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

Name	Description
condition	Required. A conditional expression.
CONTINUE **EXIT** **RETURN** *equalsStatement*	Required. The statement is executed if *condition* is true.

Notes

If the condition is true, the statement is executed.

Examples

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

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

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

IF...THEN...ELSE

Conditionally executes a statement or group of statements.
Note: Must be used inside a user-defined procedure or function.

Syntax

IF *condition* **THEN**

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

Name	Description
condition	A conditional expression.
statements	One or more statements that execute if *condition* is true.

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, _      ; first line
                    3000.0, 4000.0)             ; line continuation
```

LLOOKUP

Returns a long integer value from a lookup table.
Note: Must be used inside a user-defined procedure or function.

Syntax

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

Name	Type	Description
`result`	*long*	The returned value.
`value`	*long expression*	The lookup index.
`item0, item1, ...`	*long constant*	The list of long integer constants in the 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

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

Name	Type	Description
`result`	*float*	The value of the selected matrix element.
`row`	*long constant*	The row number of the matrix element.
`column`	*long constant*	The column number of the matrix element.

Notes

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

Examples

```
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, ...)`

Name	Type	Description
`result`	*long*	The index of the first table entry that satisfies the condition.
`value`	*long expression*	The long integer value to compare with the table items.
`cc`	*condition code*	Condition code. `Z, NZ, EQ, NE, LT, GE, LE, GT`
`item0, item1, ...`	*long constant*	The list of long integer constants for the lookup 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)`

Name	Type	Description
`value`	*long expression*	The long integer value to convert.

`format`	*long constant*	The format specifier.

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 an signed long integer. If the converted string is longer than the specified length, asterisks are stored. If the length is specified as zero, the string will be as long as necessary to represent the number. Examples of the converted string are as follows: (note: leading spaces are shown where applicable)

Value in register A	*Format byte*	*Description*	*Display format*
-1	10	signed, length = 10	`        -1`
-1	110	unsigned, length = 10	`4294967295`
-1	4	signed, length = 4	`  -1`
-1	104	unsigned, length = 4	`****`
0	4	signed, length = 4	`   0`
0	0	unformatted	`0`
1000	6	signed, length = 6	`  1000`
1000	-6	signed, length = 6, zero fill	`001000`

Examples

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

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

See Also

`FTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSEL, STRSET`
uM-FPU V3 Instruction Set: `STRINC, STRDEC`

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, ...)
```

Name	Type	Description
`result`	*long*	The index of the first table entry that satisfies the condition.
`value`	*long expression*	The long integer value to compare with the table items.
`cc`	*condition code*	Condition code. `Z, NZ, EQ, NE, LT, GE, LE, GT`
`item0,` `item1, ...`	*long constant*	The list of long integer constants for the lookup table.

Notes

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

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

```
MOP(SAVEAC)
```

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)
```

Name	Type	Description
value	*float expression*	The scalar value used for the matrix operation.

Notes

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

SCALAR_SET	Set each element of matrix A to the specified value. MA[*row, column*] = *value*
SCALAR_ADD	Add the specified value to each element of matrix A. MA[*row, column*] = MA[*row, column*] + *value*
SCALAR_SUB	Subtract the specified value from each element of matrix A. MA[*row, column*] = MA[*row, column*] - *value*
SCALAR_SUBR	Subtract the value of each element of matrix A from the specified value. MA[*row, column*] = *value* - MA[*row, column*]
SCALAR_MUL	Multiply each element of matrix A by the specified value. MA[*row, column*] = MA[*row, column*] * *value*
SCALAR_DIV	Divide each element of matrix A by the specified value. MA[*row, column*] = MA[*row, column*] / *value*
SCALAR_DIVR	Divide the specified value by each element in matrix A. MA[*row, column*] = *value* / MA[*row, column*]
SCALAR_POW	Each element of matrix A is raised to the power of the specified value. MA[*row, column*] = MA[*row, column*] ** *value*

Examples

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

Syntax

```
MOP(EWISE_SET)
MOP(EWISE_ADD)
MOP(EWISE_SUB)
MOP(EWISE_SUBR)
MOP(EWISE_MUL)
MOP(EWISE_DIV)
MOP(EWISE_DIVR)
MOP(EWISE_POW)
```

Notes

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

`EWISE_SET` Set each element of matrix A to the value of the element in matrix B.
MA[*row, column*] = MB[*row, column*]

`EWISE_ADD` Add the value of each element of matrix B to the element of matrix A.
MA[*row, column*] = MA[*row, column*] + MB[*row, column*]

`EWISE_SUB` Subtract the value of each element of matrix B from the element of matrix A.
MA[*row, column*] = MA[*row, column*] - MB[*row, column*]

`EWISE_SUBR` Subtract the value of each element of matrix A from the element of matrix B.
MA[*row, column*] = MB[*row, column*]- MA[*row, column*]

`EWISE_MUL` Multiply each element of matrix A by the element of matrix B.
MA[*row, column*] = MA[*row, column*] * MB[*row, column*]

`EWISE_DIV` Divide each element of matrix A by the element of matrix B.
MA[*row, column*] = MA[*row, column*] / MB[*row, column*]

`EWISE_DIVR` Divide each element of matrix B by the element of matrix A.
MA[*row, column*] = MB[*row, column*] / MA[*row, column*]

`EWISE_POW` Each element of matrix A is raised to the power of the element of matrix B.
MA[*row, column*] = MA[*row, column*] ** MB[*row, column*]

Examples

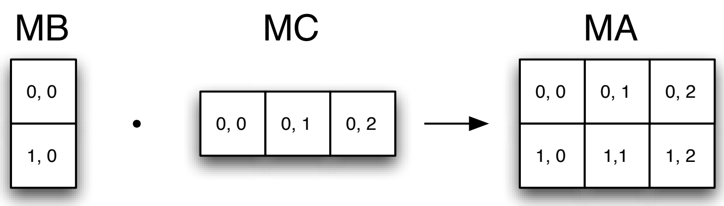
```
MOP(EWISE_DIV)      ; each elements of matrix A is divided by the element in matrix B
```

Syntax

MOP(MULTIPLY)

Notes

Performs a matrix multiplication. Matrix B is multiplied by matrix C and the result is stored in matrix A. The matrix multiply is only performed if the number of 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.



Examples

`MOP(MULTIPLY)`	; multiplies matrix A by matrix B

Syntax

MOP(IDENTITY)

Notes

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

Examples

`MOP(IDENTITY)`	; sets matrix A to the identity matrix

Syntax

MOP(DIAGONAL, value)

Name	Type	Description
`value`	*float expression*	The value to store on the diagonal.

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

Syntax

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

Name	Type	Description
return	*long* *float* *float* *float* *float*	COUNT - number of elements SUM - sum of all elements AVE - average of all elements MIN - minimum value of all elements MAX - maximum value of all elements

Notes

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

Examples

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

Syntax

```
MOP(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)`

Name	Type	Description
`return`	*float*	The determinant of matrix A.

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

MOP(LOADRA, idx1, idx2, ...)
MOP(LOADRB, idx1, idx2, ...)
MOP(LOADRC, idx1, idx2, ...)

Name	Type	Description
`idx1, idx2, ...`	*byte constants*	Index values.

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:

cos A	-sin A
sin A	cos A

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

```
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

```
MOP(LOADBA, idx1, idx2, ...)
MOP(LOADCA, idx1, idx2, ...)
```

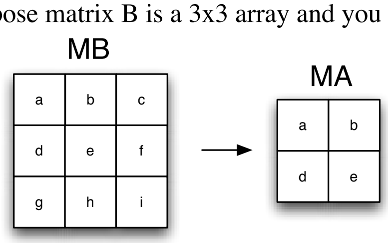
Name	Type	Description
idx1, idx2, ...	*byte constants*	Index values.

Notes

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

Examples

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




```
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, ...)
```

Name	Type	Description
idx1, idx2, ...	*byte constants*	Index values.

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:

MA

a	b	c
d	e	f
g	h	i

```
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

```
MOP(SAVEAB, idx1, idx2, ...)
MOP(SAVEAC, idx1, idx2, ...)
```

Name	Type	Description
idx1, idx2, ...	*byte constants*	Index values.

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

Syntax

```
result = POLY(value, coeff1, coeff2, ...)
```

Name	Type	Description
`result`	*float*	The result of the n^{th} order polynomial equation.
`value`	*float expression*	The value of x in the polynomial equation.
`coeff1, coeff2, ...`	*long constant*	The coefficient values the polynomial equation. Specified in order from A_N to A_0

Notes

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

$$A_0 + A_1x^1 + A_2x^2 + \ldots A_Nx^n$$

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

Examples

```
value = POLY(x, 3.0, 5.0)          ; value = 3x + 5
value = POLY(x, 1, 0, 0, 1)        ; value = x3 + 1
```

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

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

The following example makes this calculation.

```
RHlinear = POLY(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

```
result = READVAR(number)
```

Name	Type	Description
`result`	*long*	The FPU internal register value.
`number`	*byte constant*	The internal variable number. (see list below)

Notes

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

Examples

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

See Also

uM-FPU V3 Instruction Set: `READVAR`

RETURN

Returns from a user-defined procedure or function.

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

Syntax

`RETURN` `[`*`returnValue`*`]`

Name	Type	Description
`returnValue`	*long expression* *float expression*	The value returned from a user-defined function.

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, SAVEMC

Store a matrix value.

Syntax

SAVEMA(row, column, value)
SAVEMB(row, column, value)
SAVEMC(row, column, value)

Name	Type	Description
row	*long constant*	The row number of the matrix.
column	*long constant*	The column number of the matrix.
value	*float expression*	The value to store at the specified row and column.

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

```
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
```

See Also

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

SELECT...CASE

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

Syntax

```
SELECT compareItem
       statements
[CASE compareValue [, compareValue]...
       statements]...
[ELSE
       statements]
ENDSELECT
```

Name	Description
compareItem	A numeric expression or string procedure.
compareValue	A numeric or string constant.
statements	One or more statements that execute if a *compareValue* is equal to the value of *compareItem*.

Notes

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

Examples

```
n   equ   L10

SELECT n

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

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

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

ENDSELECT
```

```
n   equ   L10

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

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

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

ELSE
  n = 0                       ; otherwise, set n = 0

ENDSELECT
```

See Also

`DO...WHILE...UNTIL...LOOP`, `FOR...NEXT`, `IF...THEN`, `IF...THEN...ELSE`

SELECTMA, SELECTMB, SELECTMC

Select the registers used for matrix operations.

Syntax

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

Name	Type	Description
`reg`	*register*	The first register of the matrix.
`rows`	*long constant*	The number of rows.
`columns`	*long constant*	The number of columns.

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

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

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)

Name	Type	Description
`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)

Name	Type	Description
`string`	*string constant*	The string to send to the `SEROUT` pin.

Notes

Writes the string to the `SEROUT` pin.

Examples

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

Syntax

SERIAL(WRITE_TEXTZ, string)

Name	Type	Description
`string`	*string constant*	The string to send to the `SEROUT` pin.

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)

Name	Type	Description
value	*long expression*	The lower 8 bits of the value is output to the SEROUT pin.

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

```
SERIAL(ENABLE_CHAR)          ; enables the SERIN pin for character input
```

Syntax

SERIAL(STATUS_CHAR)

Notes

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

Examples

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

Syntax

result = **SERIAL**(READ_CHAR)

Name	Type	Description
`result`	*long*	The next available serial character value.

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

```
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

```
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

```
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

```
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)`

Name	Type	Description
`pin`	*long constant*	Output pin (0 or 1).
`action`	*long constant*	

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
```

See Also

uM-FPU V3 Instruction Set: `SETOUT`

STATUS

Checks the FPU status bits .

Syntax

STATUS(*conditionCode***)**

Name	Type	Description

conditionCode	*literal string*	A condition code symbol.

Notes

This function can only be used in a conditional expression. The STATUS condition is true if the FPU status byte agrees with if the specified 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)

Name	Type	Description
value	*long expression*	8-bit character to insert

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)

Name	Type	Description
string	*string*	A string containing the list of field separator characters.

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

FTOA, 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)

Name	Type	Description
field	*register* *long constant*	Specifies the field number.

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.

```
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 pm
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)

Name	Type	Description
string	*string*	The string to find in the string selection.

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`**`()`

Name	Type	Description
`result`	*float*	The converted value.

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.

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

See Also

`STRBYTE, STRFCHR, STRFIELD, STRFIND, STRINC, STRINS, STRLONG, STRSEL, STRSET, FTOA, LTOA`
uM-FPU V3 Instruction Set: `STRTOF`

STRINC

Increment or decrement the string selection point.

Syntax

`STRINC``(increment)`

Name	Type	Description
`increment`	*register* *long constant*	Specifies the increment or decrement amount.

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   equu  L10

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

See Also

FTOA, LTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINS, STRLONG, STRSEL, STRSET
uM-FPU V3 Instruction Set: STRINC, STRDEC

STRINS

Insert string at the string selection point.

Syntax

STRINS(string)

Name	Type	Description
string	*string*	String to insert at selection point.

Notes

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

Examples

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

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

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

`result = **STRLONG**()`

Name	Type	Description
`result`	*long*	The converted value.

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.

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

See Also

`STRBYTE, STRFCHR, 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

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

STRSEL

Set the string selection point

Syntax

`**STRSEL**([start,] length)`

Name	Type	Description
`start`	*register* *long constant*	The start of the string selection.
`length`	*long expression*	The length of the string selection.

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.

```
n   equ  L10

STRSET("0123456789ABCDEF")        ; string buffer = 0123456789ABCDEF{}
STRSEL(5, 3)                      ; string buffer = 01234{567}89ABCDEF
n = 11
STRSEL(n, 1)                      ; string buffer = 0123456789A{B}CDEF
```

See Also

FTOA, LTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSET
uM-FPU V3 Instruction Set: STRINC, STRDEC

STRSET

Copy the string to the string buffer.

Syntax

STRSET(string)

Name	Type	Description
`string`	*string*	String to store in string buffer.

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, STRINC, STRINS, STRLONG, STRSEL
uM-FPU V3 Instruction Set: STRSET

TICKLONG

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

Syntax

`result = **TICKLONG**()`

Name	Type	Description
result	*long*	The number of milliseconds since the FPU timer was started.

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**()`

Name	Type	Description
result	*long*	The number of seconds since the FPU timer was started.

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)`

Name	Type	Description
`seconds`	*long expression*	The internal seconds timer is set to this value.

Notes

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

Examples

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

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

```
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)

Name	Type	Description
reg	*register*	Register to trace.

Notes

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

Examples

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

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

```
cnt   equ L10
value equ F11
cnt = 5               ; set long integer value
value = 1.5           ; set floating point value
TRACEREG(cnt)         ; displays register 10 in debug trace
TRACEREG(value)       ; displays register 11 in debug trace
```

See Also

BREAK, TRACEOFF, TRACEON, TRACESTR
uM-FPU V3 Instruction Set: TRACEREG

TRACESTR

Display message string in the debug trace.

Syntax

TRACESTR(string)

Name	Type	Description
string	*string*	The message string.

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.

```
#FUNC 0 getID() long                      ; Function: no parameters, returns long
#FUNC % getDistance() float               ; Function: no parameters, returns float
#FUNC % getBearing(float, float) float    ; Function: two parameters, returns float
#FUNC % update                            ; Procedure: no parameters
#FUNC % getLocation(long)                 ; Procedure: one parameter
```

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.

```
#FUNC 0 addOffset(float) float            ; function prototype

#FUNCTION addOffset(float) float          ; function defintion
  return arg1 + 0.275                     ; add 0.275 to parameter 1 and return value
#END
```

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.

```
n = getID()                          ; function call
x = y + addOffset(y)                 ; function call
update                               ; procedure call
getLocation(1)                       ; procedure call
```

Nested Functions Calls

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

See Also

`#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

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

Name	Type	Description
`number`	*byte constant* `%`	Assign function to the specified EEPROM slot (0-255). Assign function to the next available EEPROM slot.
`name`	*register*	Procedure name or function name.
`arg1Type, arg2Type, ...`	*register*	Argument types. e.g. `FLOAT, LONG, ULONG`
`returnType`	*register*	Function return type. e.g. `FLOAT, LONG, ULONG`

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. *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. 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, User-defined Functions

#EEFUNCTION

Function definition for user-defined EEPROM function.

Syntax

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

Name	Type	Description
number	*byte constant*	Assign function to the specified EEPROM slot (0-255).
name	*register*	Procedure name or function name.
arg1Type, arg2Type, ...	*register*	Argument types. e.g. FLOAT, LONG, ULONG
returnType	*register*	Function return type. e.g. FLOAT, LONG, ULONG

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

```
#EEFUNC 0 getID() long              ; EEPROM function at slot 0
#EEFUNC % getDistance() float       ; EEPROM function at next available slot
#EEFUNC % getLocation(long)         ; EEPROM procedure at next available slot

#EEFUNCTION getID() long            ; EEPROM function, returns long
#EEFUNCTION getDistance() float     ; EEPROM function , returns float
#EEFUNCTION getLocation(long)       ; EEPROM procedure
```

See Also

#EEFUNC, #END, #FUNC, #FUNCTION, RETURN, User-defined Functions
uM-FPU V3 Instruction Set: EECALL, FCALL, RET, RET,cc

#END

End of user-defined function.

Syntax

`#END`

Notes

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

Examples

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

See Also

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

#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*

Name	Type	Description
`number`	*byte constant* `%`	Assign function to the specified Flash memory function (0-63). Assign function to the next available Flash memory function .
`name`	*register*	Procedure name or function name.
`arg1Type,` `arg2Type, ...`	*register*	Argument types. e.g. `FLOAT, LONG, ULONG`
`returnType`	*register*	Function return type. e.g. `FLOAT, LONG, ULONG`

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*

Name	Type	Description
reg	*register*	Register to trace.

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