uM-FPU64 IDE
Integrated Development Environment

Compiler
Release 411

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

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

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

Table of Contents

Compiler Overview

The following section provides an overview of the compiler features.

Compiling

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

Comments

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

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

Symbol Names

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

```
getDistance
latitude1
NMEA_Degrees
```

Register Data Types

The uM-FPU64 chip has 256 FPU registers. Registers 0 to 127 are 32-bit registers, and register 128 to 255 are 64-bit registers. The registers can contain any value, but the compiler requires the data type for code generation. The data types are as follows:

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

Pre-defined Register Names

The following register names are pre-defined:

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

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

User-defined Register Names

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

`x          EQU   F10`	*x* is assigned to register 10, data type is Float
`y          EQU   F%`	*y* is assigned to register 11, data type is Float
`radius     EQU   F130`	*radius* is assigned to register 130, data type is Float
`rate       EQU   L%`	*rate* is assigned to register 131, data type is Long
`radius = 1.5`	Sets *radius* to the value 1.5
`radius = radius + 0.5`	Adds the value 0.5 to *radius*

Register Bits

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

Set bit to 0:
```
register.bit = 0
```

Set bit to 1:
```
register.bit = 1
```

Toggle the bit value:
```
register.bit = ~register.bit
```

Test for bit = 0:
```
if register.bit = 0 then ...
if ~register.bit then ...
if register.bit <> 1 then ...
```

Test for bit = 1:
```
if register.bit = 1 then ...
if register.bit then ...
if register.bit <> 0 then ...
```

```
fileMode         equ    L10
READ_FILE        con    4
APPEND           con    5

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

if fileMode.READFILE then     ; execute code if bit 4 in fileMode is 1
    read(0)
endif
```

Register Arrays

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

`fval[10]  EQU  F10`	array *fval* is assigned to registers 10 to 19, data type is Float
`ma[2,3]   EQU  F%`	array *ma* is assigned to register 20 to 25, data type is Float
`val       EQU  F%`	*val* is assigned to register 26, data type is Float
`n         EQU  L%`	*n* is assigned to register 27, data type is Long
`ma[1,2] = val`	Sets the element of an array to a value
`val = fval[n]`	Stores the element of an array to register *val*

Pointers

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

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

Pointer Arrays

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

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

Register X

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

`val       EQU  F10`	*val* is assigned to register 10, data type is Float
`fval[10]  EQU  F20`	array *fval* is assigned to registers 20 to 29, data type is Float
`selectX(fval)`	set register X to point to register *fval*
`[X] = val`	stores *val* to the register pointed at by register X, then increments X
`val = [X]`	set *val* to the value of the register pointed at by register X, then increments X

Indirect Register

If a register is enclosed in square brackets, the lower 8 bits of the register value are used to reference another register.

`m          EQU  L10`	*m* is assigned to register 10, data type is Long
`[m] = val`	stores value to register pointed at by the lower 8 bits of *m*

Pointer Arithmetic

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

`P1         PTR  F20`	*P1* is assigned to register 20, the data type pointed to is Float
`P2         PTR  F%`	*P2* is assigned to register 21, the data type pointed to is Float
`n          EQU  L%`	*n* is assigned to register 23, data type is Long
`p1 = SETIND(REG_FLOAT, F40)`	set *p1* to point to register 40, data type is register, Float
`p2 = p1`	set *p2* to the value of *p1*
`p1 = p1 + 1`	increment *p1*
`p1 = p1 - 1`	decrement *p1*
`n = p2 - p1`	set *n* to the number of bytes between *p1* and *p2*

Decimal Constants

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

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

Hexadecimal Constants

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

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

Floating Point Constants

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

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

Pre-defined Constants

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

User-defined Constants

User-defined constants can be defined with the **CON** or **EQU** operator. The user-defined constant on the left of the **CON** or **EQU** operator is set to the value of the constant expression on the right. The compiler simplifies constant expressions to a single constant value. For example:
e.g.

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

or

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

String Constants

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

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

String Constant	Actual String
`"sample"`	`sample`
`"string2\r\n"`	`string2`*<carriage return><linefeed>*
`“\0C\FF”`	*binary values:* `0C, FF`
`"5\\3"`	`5\3`
`"this \"one\" "`	`this "one"`

Microcontroller Variables

Microcontroller variables are defined using the **VAR** or **EQU** operator and one of the following keywords:

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

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

When microcontroller variables are used in expressions, the IDE generates the necessary code to transfer the value between the microcontroller and the FPU. For example, the following input would generate code to load `degreesC` from the microcontroller, convert it to floating point, multiply it by 1.8, then add 32.

```
degreesC      EQU    BYTE
degreesF      EQU    F10
```

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

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

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

Operators

The following operators are supported by the compiler.

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

Operator Precedence

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

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

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

results in `F1` being set to the value of `F3` multiplied by `F4`, then added to `F2`. Parenthesis can be used to change the order of evaluation.

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

results in `F1` being set to the value of `F2` added to `F3` then multiplied by `F4`. Multiple constant values entered one after another are automatically reduced to a single constant in the expression.

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

results in `F1` being set to the value `F2` multiplied by `2.5`. If you don't want constants to be reduced, you need to use parentheses.

The code generator often adds one level of parenthesis, so parentheses in math equations should only be nested up to seven levels deep, including the parentheses used for functions.

Math Functions

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

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

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

User-Defined Functions and Procedures

User-defined functions and procedures are defined using the **#FUNCTION** directive. After a **#FUNCTION** directive is encountered, all compiled code is stored in the function specified. The end of a function occurs at the next **#FUNCTION** directive, **#END** directive, or the end of the source file. The **#FUNCTION** directive can optionally include a function name that can be used in the remainder of the source file to call the function. Function and procedure calls can be nested up to 16 levels deep.

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

```
#function 1 initDevice          user-defined procedure
  ...
#end

initDevice                      procedure call
```

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

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

tempF = convert(tempC)              function call
```

Function Prototypes

To ensure that the function being called is already defined, function prototypes can be included at the start of the program. By placing prototypes at the top of the source code, functions can be defined and called in any order, since the function values are known. Function prototypes are defined using the **FUNC** operator, which assigns a symbol name to a function number. You can assign the function number explicitly, or use the `%` character to assign the next unused function number.

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

XOP (extended opcode) Instructions

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

```
#xop quaternion:q_add       load q_add XOP
#xop quaternion:q_norm      load q_norm XOP

q_add(qa, qb, qc)           qa = qb + qc, add quaternions
tmp = q_norm(qa)            calculate the norm of quaternion qa
```

Global Symbols vs Local Symbols

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

```
tmp1   equu   F1

#function sample1
tmp2   equ    F2
```

global symbol definition

local symbol definition

```
    SELECTA, tmp1
    FSET, tmp2
#end
```

both tmp1 and tmp2 are defined inside the function

only tmp1 is defined outside the function

Control Statements

The following control statements are supported by the compiler.

```
CONTINUE

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

EXIT

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

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

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

RETURN [returnValue]

SELECT compareItem

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

```
ENDSELECT

STATUS(conditionCode)
```

Assembler Code

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

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

Wait Code

The uM-FPU64 chip has a 256 byte instruction buffer. If the instructions and data in a calculation exceed 256 bytes, the buffer could overflow, so the program must wait for the buffer to empty at least every 256 bytes. The code generated by the IDE accounts for this, and will insert a wait sequence as required. Read operations automatically generate a wait sequence, so in many applications, no additional wait sequences are required.

Summary of Statements and Functions

Control Statements

```
CONTINUE

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

EXIT

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

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

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

RETURN [returnValue]

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

STATUS(conditionCode)
```

Function Directives

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

Math Functions

```
result = SQRT(arg1)
result = LOG(arg1)
result = LOG10(arg1)
```

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

ADC Functions

```
result = ADCFLOAT(channel)
result = ADCLONG(channel)
ADCMODE(MANUAL_TRIGGER, repeat)
ADCMODE(EXTERNAL_TRIGGER, repeat)
ADCMODE(TIMER_TRIGGER, repeat, period)
ADCMODE(DISABLE)
ADCSCALE(channel, scaleFactor)
ADCTRIG
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(WRITE_FLOAT, value, format)
SERIAL(WRITE_LONG, value, format)
SERIAL(WRITE_COMMA)
SERIAL(WRITE_CRLF)
```

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

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)
DELAY(period)
```

Matrix Functions

```
FFT(type)
result = LOADMA(row, column)
result = LOADMB(row, column)
result = LOADMC(row, column)
MOP(SCALAR_SET, value)
MOP(SCALAR_ADD, value)
MOP(SCALAR_SUB, value)
MOP(SCALAR_SUBR, value)
MOP(SCALAR_MUL, value)
MOP(SCALAR_DIV, value)
MOP(SCALAR_DIVR, value)
MOP(SCALAR_POW, value)
MOP(EWISE_SET)
MOP(EWISE_ADD)
MOP(EWISE_SUB)
MOP(EWISE_SUBR)
MOP(EWISE_MUL)
MOP(EWISE_DIV)
MOP(EWISE_DIVR)
MOP(EWISE_POW)
MOP(MULTIPLY)
MOP(IDENTITY)
MOP(DIAGONAL, value)
MOP(TRANSPOSE)
return = MOP(COUNT)
```

```
return = MOP(SUM)
return = MOP(AVE)
return = MOP(MIN)
return = MOP(MAX)
MOP(COPYAB)
MOP(COPYAC)
MOP(COPYBA)
MOP(COPYBC)
MOP(COPYCA)
MOP(COPYCB)
return = MOP(DETERMINANT)
MOP(LOADRA)
MOP(LOADRB)
MOP(LOADRC)
MOP(LOADBA)
MOP(LOADCA)
MOP(SAVEAR)
MOP(SAVEAB)
MOP(SAVEAC)
SAVEMA(row, column, value)
SAVEMB(row, column, value)
SAVEMC(row, column, value)
SELECTMA(register|pointer, rows, columns)
SELECTMB(register|pointer, rows, columns)
SELECTMC(register|pointer, rows, columns)
```

Indirect Pointers

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

External Input / Output

```
DEVIO(device, action [, ...])
return = DEVIO(device, action [, ...])
DIGIO(pin+action [, ...])
return = DIGIO(pin+action [, ...])
result = EXTLONG()
EXTSET(value)
EXTWAIT
```

Miscellaneous Functions

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

Debug Functions

```
BREAK
TRACEON
TRACEOFF
TRACEREG(register)
TRACESTR(string)
```

Directives

```
#ASM
#END
#ENDASM
#FIRMWARE_REQUIRED number
#FUNC number name[(arg1Type, arg2Type, ...)]
#FUNC number name([arg1Type, arg2Type, ...]) returnType]
#FUNCTION number name[(arg1Type, arg2Type, ...)]
#FUNCTION number name([arg1Type, arg2Type, ...]) returnType]
#IDE_REQUIRED number
#TARGET_OPTIONS target, ...
```

Reference Guide: Compiler

ADCFLOAT

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

Syntax

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. (0-8)

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-FPU64 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-FPU64 Instruction Set: `ADCLONG`

ADCMODE

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

Syntax

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

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-FPU64 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 (3.3V), divided by the the number of ADC steps (the uM-FPU64 FPU has a 12-bit ADC, so there are 4095 steps).

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

See Also

`ADCFLOAT, ADCLONG, ADCMODE, ADCTRIG, ADCWAIT`
uM-FPU64 Instruction Set: `ADCSCALE`

ADCTRIG

Triggers an ADC conversion.

Syntax

`ADCTRIG`

Notes

This procedure is only required 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-FPU64 Instruction Set: `ADCTRIG`

ADCWAIT

Waits until the next ADC value is ready.

Syntax

`ADCWAIT`

Notes

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

Examples

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

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

See Also

`ADCFLOAT`, `ADCLONG`, `ADCMODE`, `ADCSCALE`, `ADCTRIG`
uM-FPU64 Instruction Set: `ADCWAIT`

BREAK

Debug breakpoint.

Syntax

BREAK

Notes

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

Examples

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

See Also

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

Conditional Expressions

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

Syntax

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

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

Examples

```
x  equ  F10
n  equ  L11

if n.5 then return

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

if n then exit

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

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

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

if status(GT) then return
```

See Also

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

CONTINUE

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

Syntax

`CONTINUE`

Notes

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

Examples

```
n  equ  L10
x  equ  F11

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

See Also

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

COPYIND

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

Syntax

COPYIND(*fromPointer*, *toPointer*, *count*)

Name	Type	Description
fromPointer	pointer	From pointer.
toPointer	pointer	To pointer.
count	long expression	The number of data items to copy.

Notes

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

Examples

```
p1 = SETIND(REG_FLOAT, F10)        ; sets pointer to register 10, data type is Float
p2 = SETIND(MEM_INT8, 100)         ; sets pointer to RAM address 100, data type is int8
COPYIND(p1, p2, 10)                ; copies 10 data values from register 10 to 19,
                                   ; converts them the data values to integer,
                                   ;stores the lower 8 bits to RAM at address 100 to109
```

See Also

SETIND, LOADIND, SAVEIND
uM-FPU64 Instruction Set: COPYIND

DELAY

Delay for the number of milliseconds.

Syntax

`DELAY(`*`period`*`)`

Name	Type	Description
period	long expression	The delay period in milliseconds.

Examples

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

See Also

`TIMESET, TICKLONG, TIMELONG`
uM-FPU64 Instruction Set: `DELAY`

DEVIO

Device Input/Output.

Syntax

Actions Supported by all Devices

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

Actions Supported by Loadable Devices

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

Device Specific Actions

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

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

DEVIO(I2C+n, START_WRITE)
DEVIO(I2C+n, STOP)
DEVIO(I2C+n, START_READ, byteCount)
```

```
DEVIO(LCD, CLEAR)
DEVIO(LCD, HOME)
DEVIO(LCD, MOVE, row, column)
DEVIO(LCD, MOVE_REG, rowRegister, columnRegister)
DEVIO(LCD, CMD, command)
DEVIO(LCD, INTERFACE, type)
DEVIO(LCD, BACKLIGHT_ON)
DEVIO(LCD, BACKLIGHT_OFF)

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

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

DEVIO(SERVO+n, PULSE, register)
DEVIO(SERVO+n, SPEED, register)
DEVIO(SERVO+n, TIME, register)
DEVIO(SERVO+n, MOVE, register)
DEVIO(SERVO+n, HOME, register)
result = DEVIO(SERVO+n, READ_PULSE)
result = DEVIO(SERVO+n, STATUS)

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

Name	Type	Description
device	pre-defined symbol and byte constant	The device type and number. FIFO1, FIFO2, FIFO3, FIFO4, OWIRE, I2C, SPI, ASYNC, COUNTER, SERVO, LCD, VDRIVE2
action	pre-defined symbols	The device action and modifiers.

Notes

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

Examples

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

See Also

uM-FPU64 Instruction Set: DEVIO

DIGIO

Set the `OUT0` or `OUT1` output pin.

Syntax

```
DIGIO(pin+LOW)
DIGIO(pin+HIGH)
DIGIO(pin+TOGGLE)
DIGIO(pin+INPUT)
DIGIO(pin+WRITE_BITS, bitCount[+PRE][+POST][+MSB][+LSB][+FAST][+SLOW], value)
result = DIGIO(pin+READ_BITS, bitCount[+PRE][+POST][+MSB][+LSB][+FAST][+SLOW])
DIGIO(pin+WRITE_BITP, bitCount[+INVERT], bitValue)
result = DIGIO(pin+READ_BITP, bitCount[+INVERT])
```

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

Notes

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

Examples

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

See Also

uM-FPU64 Instruction Set: `DIGIO`

DO...WHILE...UNTIL...LOOP

Repeatedly execute a group of statements while specified conditions are true.
Note: Must be used inside a user-defined procedure or function.

Syntax

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

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

EVENT

Manage background events.

Syntax

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

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

Notes

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

Examples

```
EVENT(ENABLE+RTC)      ; enable RTC event
```

See Also

uM-FPU64 Instruction Set: `EVENT`

EXIT

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

Syntax

`EXIT`

Notes

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

Examples

```
n   equ  L10
x   equ  F11

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

See Also

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

Expressions

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

The math operators are as follows:

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

<table>
<tr><th>Operator Precedence</th></tr>
<tr><td>~ + -</td></tr>
<tr><td>**</td></tr>
<tr><td>* / %</td></tr>
<tr><td>+ -</td></tr>
<tr><td><< >></td></tr>
<tr><td>&</td></tr>
<tr><td>^</td></tr>
<tr><td>|</td></tr>
</table>

Examples

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

See Also

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

EXTLONG

Returns the value of the external input counter.

Syntax

`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-FPU64 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-FPU64 Instruction Set: `EXTSET`

EXTWAIT

Wait for the next external input to occur.

Syntax

`EXTWAIT`

Notes

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

Examples

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

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

See Also

`EXTLONG, EXTSET`
uM-FPU64 Instruction Set: `EXTWAIT`

FCNV

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

Syntax

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-FPU64 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-FPU64 Instruction Set: `FFT`

FLOOKUP

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

Syntax

`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. A maximum of 256 values are allowed.

Notes

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

Examples

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

See Also

`FTABLE, LLOOKUP, LTABLE`
uM-FPU64 Instruction Set: `TABLE`

FOR...NEXT

Executes a group of statements a specified number of times.
Note: Must be used inside a user-defined procedure or function.

Syntax

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

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. A maximum of 256 values are allowed.

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-FPU64 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-FPU64 Instruction Set: STRINC, STRDEC

IF...THEN

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

Syntax

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

Name	Description
condition	A conditional expression.
CONTINUE **EXIT** **RETURN** *equalsStatement*	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. A maximum of 256 values are allowed.

Notes

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

Examples

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

See Also

`FLOOKUP, FTABLE, LTABLE`
uM-FPU64 Instruction Set: `TABLE`

LOADMA
LOADMB
LOADMC

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

Syntax

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

Name	Type	Description
result	float	The value of the selected matrix element.
row	long constant | register	The row number of the matrix element, or a register containing the row number.
column	long constant | register	The column number of the matrix element, or a register containing the column number.

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-FPU64 Instruction Set: `LOADMA, LOADMB, LOADMC`

LTABLE

Returns the index of the first table entry that satisfies the condition code. The specified value is compared to each value in the list of items, and the index value is returned. The index value starts at zero.
Note: Must be used inside a user-defined procedure or function.

Syntax

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

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-FPU64 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 *format* is zero, the length of the converted string is variable and can range from 1 to 11 characters in length. Examples of the converted string are as follows:

```
1
500000
-3598390
```

If *format* is non-zero, a value between 0 and 15 specifies the length of the converted string. The converted string is right justified. If *format* is positive, leading spaces are used. If *format* 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-FPU64 Instruction Set: STRINC, STRDEC

Math Functions

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

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

Examples

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

See Also

uM-FPU64 Instruction Set: Each of the functions uses an FPU instruction of the same name (`ABS`, `MOD`, `MIN` and `MAX` use the `FABS`, `FMOD`, `FMIN`, `FMAX` instructions for floating point data types, and the `LABS`, `LDIV` (remainder), `LMIN`, `LMAX` instructions for Long or Unsigned data types).

MOP

Performs matrix operations. The matrix operations are summarized below.

```
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(COPY_AB)
MOP(COPY_AC)
MOP(COPY_BA)
MOP(COPY_BC)
MOP(COPY_CA)
MOP(COPY_CB)
return = MOP(DETERMINANT)
MOP(INVERSE)
MOP(LOAD_RA, idx1, idx2, ...)
MOP(LOAD_RB, idx1, idx2, ...)
MOP(LOAD_RC, idx1, idx2, ...)
MOP(LOAD_BA, idx1, idx2, ...)
MOP(LOAD_CA, idx1, idx2, ...)
MOP(SAVE_AR, idx1, idx2, ...)
MOP(SAVE_AB, idx1, idx2, ...)
MOP(SAVE_AC, idx1, idx2, ...)
MOP(LU_DECOMP)
MOP(LU_INVERSE)
MOP(LU_DETERM)
MOP(LU_SOLVE)
MOP(CH_DECOMP)
MOP(CH_INVERSE)
MOP(CH_DETERM)
MOP(CH_SOLVE)
```

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

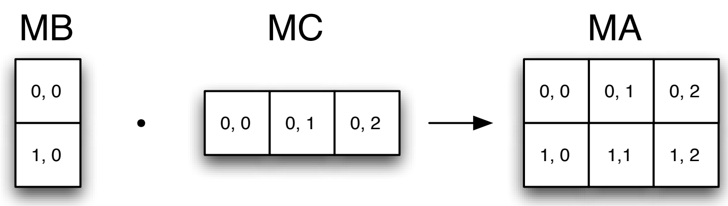
```
MOP(EWISE_DIV)          ; each elements of matrix A is divided by the element in matrix B
```

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



Examples

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

Notes

Copies one matrix to another.

Examples

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

Syntax

```
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(LOAD_RA, idx1, idx2, ...)
MOP(LOAD_RB, idx1, idx2, ...)
MOP(LOAD_RC, idx1, idx2, ...)
```

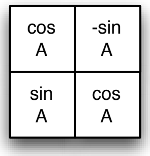
Name	Type	Description
idx1, *idx2*, ...	byte constants	Index values.

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:



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(LOAD_RA, 2, -1 , 1, 2) ; sets matrix A to the rotation matrix shown above
```

Syntax

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

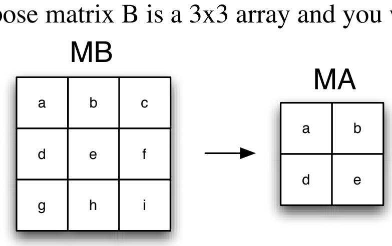
Name	Type	Description
idx1, *idx2*, ...	byte constants	Index values.

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

Syntax

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

Syntax

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

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.

Syntax

MOP(LU_DECOMP)
MOP(LU_INVERSE)

```
MOP(LU_DETERM)
MOP(LU_SOLVE)
MOP(CH_DECOMP)
MOP(CH_INVERSE)
MOP(CH_DETERM)
MOP(CH_SOLVE)
```

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

See Also

LOADMA, LOADMB, LOADMC, SAVEMA, SAVEMB, SAVEMC, SELECTMA, SELECTMB, SELECTMC
uM-FPU64 Instruction Set: MOP

POLY

Calculates the n^{th} order polynomial of the floating point value.
Note: Must be used inside a user-defined procedure or function.

Syntax

```
result = 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-FPU64 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-FPU64 Instruction Set: `READVAR`

RETURN

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

Syntax

RETURN [*returnValue*]

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

RTC

Manage the real-time clock.

Syntax

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

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

Notes

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

Examples

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

See Also

uM-FPU64 Instruction Set: `RTC`

SAVEMA
SAVEMB
SAVEMC

Store a matrix value.

Syntax

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

Name	Type	Description
row	long constant | register	The row number of the matrix element, or a register containing the row number.
column	long constant | register	The column number of the matrix element, or a register containing the column number.
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-FPU64 Instruction Set: SAVEMA, SAVEMB, SAVEMC

SELECT...CASE

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

Syntax

```
SELECT compareItem

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

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   equu  L10

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

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

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

ELSE
  n = 0                       ; otherwise, set n = 0

ENDSELECT
```

See Also

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

SELECTA

Select register A.

Syntax

SELECTA(*register*)

Name	Type	Description
register	register	The register to select as register A.

Notes

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

Examples

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

See Also

SELECTX
uM-FPU64 Instruction Set: SELECTA

SELECTMA
SELECTMB
SELECTMC

Select the registers used for matrix operations.

Syntax

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

Name	Type	Description
register	register	The first register of the matrix.
pointer	pointer	Pointer to first element of the matrix.
row	long constant | register	The number of rows, or a register containing the number of rows.
column	long constant | register	The number of columns, or a register containing the number of columns.

Notes

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

Examples

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

See Also

MOP, LOADMA, LOADMB, LOADMC, SAVEMA, SAVEMB, SAVEMC
uM-FPU64 Instruction Set: SELECTMA, SELECTMB, SELECTMC

SELECTX

Select register X.

Syntax

`SELECTA``(`*`register`*`)`

Name	Type	Description
`register`	register	The register to select as register X.

Notes

This procedure is used to set register X prior to using the **`[X]`** operator.

Examples

```
SELECTX(F20)            ; set register X to point to register 20

[X] = 10.5              ; stores 10.5 to the register pointed to by register X, then increments X
```

See Also

`SELECTA`
uM-FPU64 Instruction Set: `SELECTX`

SERIAL

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

```
SERIAL(SET_BAUD, baud)
SERIAL(WRITE_STR, string)
SERIAL(WRITE_STRZ, string)
SERIAL(WRITE_SBUF)
SERIAL(WRITE_SSEL)
SERIAL(WRITE_CHAR, value)
SERIAL(WRITE_FLOAT, value, format)
SERIAL(WRITE_LONG, value, format)
SERIAL(WRITE_COMMA)
SERIAL(WRITE_CRLF)

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

See Also

uM-FPU64 Instruction Set: `SEROUT, SERIN`

A detailed description of each `SERIAL` operation is shown below.

Syntax

SERIAL(SET_BAUD, baud)

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_STR, *string*)

Name	Type	Description

string	string constant	The string to send to the specified serial output.

Notes
Writes the string to the specified serial output.

Examples

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

Syntax
SERIAL(WRITE_STRZ, *string*)

Name	Type	Description
string	string constant	The string to send to the specified serial output.

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

Examples

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

Syntax
SERIAL(WRITE_SBUF)

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

Examples

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

Syntax
SERIAL(WRITE_STRSEL)

Notes
Writes the current string selection to the specified serial output.

Examples

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

Syntax

SERIAL(WRITE_CHAR, *value*)

Name	Type	Description
value	long expression	The lower 8 bits are output to the specified serial output.

Notes

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

Examples

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

Syntax

SERIAL(WRITE_FLOAT, *value*, *format*)

Name	Type	Description
value	float expression	The floating point value to convert.

Notes

Converts the *value* to a floating point string with the specified *format*, and writes the string to the specified serial output.

Examples

```
SERIAL(WRITE_FLOAT, pi, 42)         ; sends 3.14 to the SEROUT pin
SERIAL(WRITE_FLOAT, value, 0)       ; sends floating point string to the SEROUT pin
```

Syntax

SERIAL(WRITE_LONG, *value*, *format*)

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

Notes

Converts the *value* to a floating point string with the specified *format*, and writes the string to the specified serial output.

Examples

```
SERIAL(WRITE_FLOAT, pi, 42)         ; sends 3.14 to the serial output
SERIAL(WRITE_FLOAT, value, 0)       ; sends floating point string to the SEROUT pin
```

Syntax
SERIAL(WRITE_COMMA)

Notes
Writes a comma to the specified serial output.

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

Syntax
SERIAL(WRITE_CRLF)

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

Examples
```
SERIAL(WRITE_CRLF)                     ; sends CR/LF 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` or `ASYNC` pin is enabled for character input. Received characters are stored in a 160 byte input buffer. The serial input status can be checked with the `SERIAL(STATUS_CHAR)` procedure and characters can be read using the `SERIAL(READ_CHAR)` function.

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

Syntax

`SERIAL(STATUS_CHAR)`

Notes

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

Examples

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

Syntax

result = **SERIAL**(READ_CHAR)

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.

Examples

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

Syntax

`SERIAL(ENABLE_NMEA)`

Notes

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

Examples

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

Syntax

SERIAL(STATUS_NMEA)

Notes

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

Examples

```
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
```

SETIND

Return a pointer to a register or memory location.

Syntax

```
SETIND(type, reg)
SETIND(type, address)
SETIND(type, function, offset)
```

Name	Type	Description
type	*symbol*	Specifies data type of the pointer. REG_LONG, REG_FLOAT, FLASH_UINT8, FLASH_INT16, MEM_UINT16, FLASH_LONG32, FLASH_FLOAT32, FLASH_LONG64, FLASH_FLOAT64, DMA_UINT8, DMA_INT16, DMA_UINT16, DMA_LONG32, DMA_FLOAT32, DMA_LONG64, DMA_FLOAT64, FLASH_UINT8, FLASH_INT16, FLASH_UINT16, FLASH_LONG32, FLASH_FLOAT32, FLASH_LONG64, FLASH_FLOAT64
reg	*register*	The register number for the pointer. Required for REG_ data types.
address	*long constant*	The memory address for the pointer. Required for MEM_ and DMA_ data types.
function	*function*	The function number for the pointer. Required for FLASH_ data types.
offset	*long constant*	The function offset for the pointer. Required for FLASH_ data types.

Notes

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

Examples

```
p = SETIND(REG_FLOAT, F10)          ; sets pointer to register 10, data type is Float
p = SETIND(MEM_INT8, 100)           ; sets pointer to RAM address 100, data type is int8
p = SETIND(FLASH_FLOAT32, 0, 0)     ; sets pointer to Flash function 0, offset 0
```

See Also

COPYIND
uM-FPU64 Instruction Set: SETIND

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 *conditionCode*. If the NMEA_CHECKSUM or NMEA_OVERRUN *conditionCode* 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, {} characters are used to shown the string selection point.

```
n   equ   L10

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

See Also

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

uM-FPU64 Instruction Set: STRBYTE

STRFCHR

Sets the field separator characters used by the `STRFIELD` procedure.

Syntax

`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-FPU64 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. `STRFIELD` can also be used in a conditional expression.

Examples

The following example shows how a date/time string can be parsed.
Note: In the following example the {} characters are used to shown the string selection point.

```
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

if strfield() = “GPRMC” then      ; check for GPRMC sentence
 …
endif
```

See Also

`FTOA, LTOA, STRBYTE, STRFCHR, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSEL, STRSET`
uM-FPU64 Instruction Set: `STRINC, STRDEC`

STRFIND

Find the string in the current string selection.

Syntax

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-FPU64 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 *result*. Conversion stops at the first character that is not a valid character for a floating point number.

Examples

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

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

See Also

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

uM-FPU64 Instruction Set: STRTOF

STRINC

Increment or decrement the string selection point.

Syntax

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  equ  L10

STRSET("abcdef")            ; string buffer = abcdef{}
STRSEL(0,127)               ; string buffer = {abcdef}
STRFIND("d")                ; string buffer = abc{d}ef
STRINC(-2)                  ; string buffer = a{}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-FPU64 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-FPU64 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 *result*. Conversion stops at the first character that is not a valid character for a long integer number.

Examples

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

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

See Also

STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS, STRSEL, STRSET, FTOA, LTOA
uM-FPU64 Instruction Set: STRTOL

String Constant

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

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

Examples

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. STRSEL can also be used in a conditional expression.

Examples

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

```
n   equ  L10

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

if STRSEL(2,2) = “W1” then        ; check if selection = “W1”
 …
endif
```

See Also

FTOA, LTOA, STRBYTE, STRFCHR, STRFIELD, STRFIND, STRFLOAT, STRINC, STRINS, STRLONG, STRSET
uM-FPU64 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 buffer.

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-FPU64 Instruction Set: STRSET

TICKLONG

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

Syntax

result = **TICKLONG**()

Name	Type	Description
result	long	The number of milliseconds or microseconds 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 timer is a 32-bit register.

Examples

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

See Also

TIMELONG, TIMESET
uM-FPU64 Instruction Set: TICKLONG

TIMELONG

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

Syntax

result = **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 seconds timer is a 32-bit register.

Examples

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

See Also

`TICKLONG, TIMESET`
uM-FPU64 Instruction Set: `TIMELONG`

TIMESET

Set internal timer values.

Syntax

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 *seconds* 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-FPU64 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-FPU64 Instruction Set: `TRACEON, TRACEOFF`

TRACEREG

Display register value in the debug trace.

Syntax

TRACEREG(*register*)

Name	Type	Description
register	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-FPU64 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 *string* is displayed in the debug trace window. If the debugger is disabled, this procedure is ignored.

Examples

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

`"test1"`

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

See Also

`BREAK, TRACEOFF, TRACEON, TRACEREG`

uM-FPU64 Instruction Set: `TRACESTR`

User-defined Functions

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

Defining Functions

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

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

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

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

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

```
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

`#END, #FUNC, #FUNCTION`
uM-FPU64 Instruction Set: `FCALL, RET, RET,cc`

XOP (extended opcode) Instructions

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

Defining XOPs

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

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

Passing Arguments to the Quaternion XOPs

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

```
q_add(qa, qb, qc)         ; add quaternion add XOP
tmp = q_norm(qa)          ; calculate the norm of the quaternion
```

#ASM

Start of assembler code.

Syntax

`#ASM`

Notes

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

Examples

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

See Also

`#ENDASM`

#DEVICE

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

Syntax

#XOP `device_file{:device_name}`

Name	Type	Description
`device_file`	*string*	Specifies the name of a Device Library File.
`device_name`	*string*	Specifies the name of the loadable device. If *device_name* is not specified, then *device_name* is the same as the *device_file* name.

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

Examples

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

#END

End of user-defined function.

Syntax

`#END`

Notes

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

Examples

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

See Also

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

#ENDASM

End of assembler code.

Syntax

`#ENDASM`

Notes

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

Examples

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

See Also

`#ENDASM`

#FIRMWARE_REQUIRED

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

Syntax

#FIRMWARE_REQUIRED, *release*

Name	Type	Description
release	*number*	Four digit uM-FPU64 release code.

Examples

```
#FIRMWARE_REQUIRED 4050
```

#FUNC

Prototype for user-defined function stored in Flash memory.

Syntax

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

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

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

Examples

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

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

See Also

#END, #FUNC, RETURN, *User-defined Functions*
uM-FPU64 Instruction Set: FCALL, RET, RET,cc

#IDE_REQUIRED

Specifies the minimum IDE release required for the code.

Syntax

#IDE_REQUIRED, *release*

Name	Type	Description
release	*number*	Three digit uM-FPU64 IDE release number.

Examples

```
#IDE_REQUIRED 407
```

#TARGET_CODE

Specify target code link.

Syntax

#TARGET_CODE *link_ID*

Name	Type	Description
link_ID	*string*	Specifies a unique link ID.

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

Example

This source code:

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

Generates the following output for the Arduino target:

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

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

#TARGET_OPTIONS

Specify target options.

Syntax

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

Name	Type	Description
`X|M2|X1|X2`	*string*	Specifies the type of PICAXE chip used.
`B0...B55`	*register*	Specifies the PICAXE variable used by the FPU support routines.

PICAXE

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

PICMODE

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

PROPELLER

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

Examples

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

#XOP

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

Syntax

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

Name	Type	Description
`xop_file`	*string*	Specifies the name of an XOP Library File.
`xop_name`	*string*	Specifies the name of an XOP instruction. If *xop_name* is not specified, then *xop_name* is the same as the *xop_file* name.

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

Examples

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

Reference Guide: Assembler

Assembler code can be entered by enclosing it with the **#ASM** and **#ENDASM** directives. Multiple instructions can be entered on a single line, and an instruction can span more than one line, but each element of an instruction (e.g. a number or string) must be on a single line. For example:

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

single line of assembler

or

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

multiple lines of assembler

Assembler Instructions

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

```
NOP
SELECTA, reg
SELECTX, reg
CLR, reg
CLRA
CLRX
CLR0
COPY, reg, reg
COPYA, reg
COPYX, reg
LOAD, reg
LOADA
LOADX
ALOADX
XSAVE, reg
XSAVEA
COPY0, reg
LCOPYI, bb, reg
SWAP, reg, reg
SWAPA, reg
LEFT
RIGHT
FWRITE, reg, floatval
FWRITEA, floatval
FWRITEX, floatval
FWRITE0, floatval
FREAD
FREADA
FREADX
FREAD0
ATOF, string
FTOA, bb
FSET, reg
FADD, reg
FSUB, reg
FSUBR, reg
FMUL, reg
FDIV, reg
FDIVR, reg
FPOW, reg
FCMP, reg
FSET0
FADD0
FSUB0
FSUBR0
FMUL0
FDIV0
FDIVR0
FPOW0
FCMP0
FSETI, bb
FADDI, bb
FSUBI, bb
FSUBRI, bb
FMULI, bb
FDIVI, bb
FDIVRI, bb
FPOWI, bb
FCMPI, bb
FSTATUS, reg
FSTATUSA
FCMP2, reg, reg
FNEG
FABS
FINV
SQRT
ROOT, reg
LOG
LOG10
EXP
EXP10
SIN
COS
TAN
ASIN
ACOS
ATAN
ATAN2, reg
DEGREES
RADIANS
FMOD, reg
FLOOR
CEIL
ROUND
FMIN, reg
FMAX, reg
FCNV, bb
FMAC, reg, reg
FMSC, reg, reg
LOADBYTE bb
LOADUBYTE bb
LOADWORD wwww
LOADUWORD wwww
LOADE
LOADPI
FCOPYI, bb, reg
FLOAT
FIX
FIXR
FRAC
FSPLIT
SELECTMA, reg, bb, bb
```

```
SELECTMB, reg, bb, bb
SELECTMC, reg, bb, bb
LOADMA, bb, bb
LOADMB, bb, bb
LOADMC, bb, bb
SAVEMA, bb, bb
SAVEMB, bb, bb
SAVEMC, bb, bb
MOP, bb
FFT, bb
WRIND, bb, bb...
RDIND, bb
DWRITE, reg, float64val
DREAD, reg
LBIT, bb, reg
SETIND, bb, bb
ADDIND, reg, bb
COPYIND, reg, reg, reg
LOADIND, reg
SAVEIND, reg
INDA, reg
INDX, reg
FCALL, fnum
EVENT, bb
RET
BRA, _label
BRA, cc, _label
JMP, _label
JMP, cc, _label
TABLE, bb
FTABLE, bb
LTABLE, bb
POLY, bb
GOTO, reg
LWRITE, reg, longval
LWRITEA, longval
LWRITEX, longval
LWRITE0, longval
LREAD
LREADA
LREADX
LREAD0
LREADBYTE
LREADWORD
ATOL, string
LTOA, bb
LSET, reg
LADD, reg
LSUB, reg
LMUL, reg
LDIV, reg
LCMP, reg
LUDIV, reg
LUCMP, reg
LTST, reg
LSET0
LADD0
LSUB0
LMUL0
LDIV0
LCMP0
LUDIV0
LUCMP0
LTST0
LSETI, bb
LADDI, bb
LSUBI, bb
LMULI, bb
LDIVI, bb
LCMPI, bb
LUDIVI, bb
LUCMPI, bb
LTSTI, bb
LSTATUS, reg
LSTATUSA
LCMP2, reg, reg
LUCMP2, reg, reg
LNEG
LABS
LINC, reg
LDEC, reg
LNOT
LAND, reg
LOR, reg
LXOR, reg
LSHIFT, reg
LMIN, reg
LMAX, reg
LONGBYTE, bb
LONGUBYTE, bb
LONGWORD, wwww
LONGUWORD, wwww
LSHIFTI, bb
LANDI, bb
LORI, bb
DIGIO, bb
ADCMODE, bb
ADCTRIG
ADCSCALE, bb
ADCLONG, bb
ADCLOAD, bb
ADCWAIT
TIMESET
TIMELONG
TICKLONG
DEVIO, dev, bb...
DELAY, wwww
RTC, bb
SETARGS
EXTSET
EXTLONG
EXTWAIT
STRSET, string
STRSEL, bb, bb
STRINS, string
STRCMP, string
STRFIND, string
STRFCHR, string
STRFIELD, bb
STRTOF
STRTOL
READSEL
SYNC
READSTATUS
READSTR
VERSION
IEEEMODE
PICMODE
CHECKSUM
BREAK
TRACEOFF
TRACEON
TRACESTR, string
TRACEREG, reg
READVAR, bb
SETREAD
RESET
```

Where:

`reg`	register number (0-127)
`fnum`	Flash function number (0-63)
`bb`	8-bit value
bb…	multiple 8-biy values
dev	device
wwww	16-bit value
`_label`	address label
`cc`	condition code `(Z,EQ,NZ,NE,LT,LE,GT,GE,PZ,MZ,INF,FIN,PINF,MINF,NAN,TRUE,FALSE)`
`floatval`	floating point value
`longval`	long integer value

`string`	ASCII string

Data Directives

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

`#BYTE`	8-bit data values
`#WORD`	6-bit integer values
`#LONG, #LONG32`	32-bit integer values
`#LONG64`	64-bit integer values
`#FLOAT, #FLOAT32`	32-bit floating point values
`#DOUBLE, #FLOAT64`	64-bit floating point values

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

POLY instruction with coefficients -0.0000028, 0.0405, -4.0

```
#byte, 3, -3, 5, 4
```

defines three 8-bit integer values

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

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

Symbol Definitions

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

```
angle  EQU    F10

#asm
   SELECTA, angle
#endasm
```

symbol definition

symbol used by assembler instruction

Branch and Return Instructions

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

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

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

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

Condition Codes

The condition codes used by various instructions are summarized below.

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

Labels

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

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

Using Branch Instructions and Labels

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

If Statement

Psuedocode

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

end if

Assembler Code

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

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

Repeat Statement

Psuedocode

repeat 10 times
    sum = sum + 1

Assembler Code

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

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

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

For Statement

Psuedocode

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

Assembler Code

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

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

    LINC, cnt                   increment loop counter
    LCMP2, cnt, endValue        check for end value
```

```
    BRA, LT, _loop
#endasm
```

repeat until done

String Arguments

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

```
STRSET, "test"
```

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

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

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

add carriage return, linefeed between line1 and line2

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

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

results in the same string as above

An empty string can be specified in two ways.

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

empty string

Table Instructions

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

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

load value from table

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

POLY instruction with coefficients -0.0000028, 0.0405, -4.0

MOP Instruction

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

```
#asm
```

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

See the *uM-FPU64 Instruction Set* document for a list of the predefined symbols for matrix operations.

Reference Guide: Target Description File

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

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

User target files are loaded from the following folder:

\My Documents\Micromega\Target Files

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

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

A sample target description file is shown below.

```
TARGET_NAME=<Generic C compiler>

; This file defines code generation for a C compiler

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

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

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

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

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

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

Syntax

The general format of a command is as follows:

```
COMMAND=<ARGUMENT>
```

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

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

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

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

Tab Spacing

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

Commands

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

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

START_WRITE_TRANSFER
START_READ_TRANSFER
STOP_TRANSFER
WAIT

WRITE
WRITE_BYTE_FORMAT
WRITE_CMD
WRITE_DATA
WRITE_WORD_FORMAT
WRITE_LONG_FORMAT
WRITE_FLOAT_FORMAT
WRITE_STRING_FORMAT

WRITE_BYTE
```

```
WRITE_WORD
WRITE_LONG
WRITE_STRING

READ_CMD
READ_DELAY
READ_BYTE
READ_WORD
READ_LONG
READ_FLOAT

REGISTER_DEFINITION
BYTE_DEFINITION
WORD_DEFINITION
LONG_DEFINITION
FLOAT_DEFINITION

PRINT_FLOAT
PRINT_LONG
PRINT_FPUSTRING
PRINT_NEWLINE
PRINT_STRING

RESERVED_PREFIX
RESERVED_WORD
```

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

Reviewing the Sample File

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

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

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

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

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

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

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

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

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

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

```
WRITE_WORD=<{t}fpu_writeWord({word});>
WRITE_LONG=<{t}fpu_writeLong({long});>
WRITE_FLOAT=<{t}fpu_writeFloat({float});>
WRITE_STRING=<{t}fpu_writeChar("{string}");>
```

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

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

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

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

The commands for reading data values are shown below.

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

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

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

The following command specifies how registers are defined .

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

An example of register definitions is as follows:

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

Next are the commands to define microcontroller variable.

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

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

```
int cnt;
```

```
long sum;
float angle;
```

Finally, the commands to define print statement.

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

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

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

Reserved Words

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

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

An example of the opcodes generated is as follows:

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

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

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

An example of the opcodes generated is as follows:

```
SELECTA
FADD
F_SIN
F_COS
```

Target Description Commands

BYTE_DEFINITION — **Define byte variable definition**

`BYTE_DEFINITION=<string>`

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

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

COMMENT_PREFIX — **Set the prefix for comments**

`COMMENT_PREFIX=<string>`

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

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

CONTINUATION — **Define line continuation for WRITE command**

`CONTINUATION=<string>`

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

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

DECIMAL_FORMAT — **Set the prefix for decimal numbers**

`DECIMAL_FORMAT=<string>`

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

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

FLOAT_DEFINITION — Define float variable definition

`FLOAT_DEFINITION=<string>`

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

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

HEX_FORMAT — Set the prefix for hexadecimal numbers

`HEX_FORMAT=<string>`

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

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

LONG_DEFINITION — Define long variable definition

`LONG_DEFINITION=<string>`

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

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

MAX_LENGTH — Set maximum length of write instruction

`MAX_LENGTH=<length>`

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

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

MAX_WRITE — Set maximum number of bytes in write instruction

`MAX_WRITE=<n>`

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

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

OPCODE_PREFIX **Set the prefix for opcodes in WRITE command**

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

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

PRINT_FLOAT **Define instructions to print float value**

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

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

PRINT_FPUSTRING **Define instructions to print FPU string**

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

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

PRINT_LONG **Define instructions to print long value**

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

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

PRINT_NEWLINE **Define instructions to print new line**

`PRINT_NEWLINE=<string>`

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

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

PRINT_STRING **Define instructions to print text string**

`PRINT_STRING=<string>`

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

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

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

`READ_BYTE=<string>`

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

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

READ_CMD **Define format for Read Command Instructions**

`READ_CMD=<string>`

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

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

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

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

READ_DELAY **Define instructions for read delay**

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

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

READ_LONG — Defines command to read 32-bit value

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

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

READ_WORD — Defines instructions to read 16-bit value

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

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

REGISTER_DEFINITION — Define register definition

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

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

RESERVED_PREFIX — Define prefix for reserved words

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

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

RESERVED_WORD **Define reserved word**

RESERVED_WORD=<string>

Default: empty string

Parameters: none

Example: RESERVED_WORD=<SIN>

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

SEPARATOR **Define separator character for WRITE command**

SEPARATOR=<string>

Default: , (comma and space)

Parameters: none

Example: SEPARATOR=<, >

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

SOURCE_PREFIX **Set indent for the start of a comment line**

SOURCE_PREFIX=<string>

Default: ; (semi-colon)

Parameters: none

Example: SOURCE_PREFIX=<   ;-->

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

START_READ_TRANSFER **Define instructions for start of a read transfer**

START_READ_TRANSFER=<string>

Default: empty string

Parameters: none

Example: START_READ=<CALL START_READ();>

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

START_WRITE_TRANSFER **Define instructions for start of a write transfer**

START_WRITE_TRANSFER=<string>

Default: empty string

Parameters: none
Example: START_WRITE=<CALL START_WRITE();>

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

STOP_TRANSFER **Define instructions for end of read or write transfer**

STOP_TRANSFER=<string>
Default: empty string
Parameters: none
Example: STOP=<CALL STOP();>

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

STRING_HEX_FORMAT **Define format for non-printable string characters**

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

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

TAB_SPACING **Set number of characters per tab**

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

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

TARGET_NAME **Define the target name**

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

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

TARGET_OPTIONS **Define target options**

`TARGET_OPTIONS=<target, ...>`

Default: none
Parameters: device specific
Example: `TARGET_OPTIONS=<PICAXE, X2>`

Description: The target options are device specific.

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

{X | M2 | X1 | X2} specifies the type of PICAXE chip used
{B0...B55} specifies the PICAXE variable used by the FPU support routines

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

`TARGET_OPTIONS=<PICMODE>`

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

`TARGET_OPTIONS=<PROPELLER>`

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

WAIT **Define instructions to wait for ready status**

`WAIT=<string>`

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

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

WORD_DEFINITION **Define word variable definition**

`WORD_DEFINITION=<string>`

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

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

WRITE **Define instructions to write bytes**

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

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

WRITE_BYTE **Define instructions to write 8-bit value**

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

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

WRITE_BYTE_FORMAT **Define 8-bit value format for WRITE command**

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

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

WRITE_CMD **Define format for Write Command Instructions**

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

Description: This command defines the format of the write command instructions. Parameter {n} is replaced with the appropriate write command suffix as follows:

WriteCmd, WriteCmdByte, WriteCmdByte2, WriteCmdByte3,
WriteCmdByte4, WriteCmdByteWord, WriteCmdByte2Word,
WriteCmdByte2Word, WriteCmdByteLong, WriteCmdWord,
WriteCmdLong, WriteCmdStr, WriteCmdByteStr, WriteCmdByte2Str.

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

WRITE_DATA **Define format for Write Data Instructions**

WRITE_DATA=<string>

Default: empty string

Parameters: {n},{data}

Example: WRITE_DATA=<FPU.WriteData{n}({data}>

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

WriteData1, WriteData2, WriteData3, …

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

00	8-bit data
01	16-bit data
10	32-bit data
11	8-bit opcode

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

WRITE_LONG **Define instructions to write 32-bit value**

WRITE_LONG=<string>

Default: empty string

Parameters: none

Example: WRITE_LONG=<call fpu_writelong({long});>

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

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

WRITE_LONG_FORMAT=<string>

Default: empty string

Parameters: none

Examples:
```
WRITE_LONG=<{byte}<<24, {byte}<<16, {byte}<<8, {byte}>
WRITE_LONG=<{word}(1), {word}(2)>
WRITE_LONG=<{long}>
```

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

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

`WRITE_WORD=<string>`

Default: empty string

Parameters: none

Example: `WRITE_WORD=<call fpu_writeWord{word});>`

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

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

`WRITE_WORD_FORMAT=<string>`

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

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

WRITE_STRING **Define instructions to write string value**

`WRITE_STRING=<string>`

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

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

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

`WRITE_STRING_FORMAT=<string>`

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

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