SmplMath

Macros-system for MASM and jWasm, allowing to use mathematic expression as known from high level programming language like c/c++,…

by ***qWord*** – contactable through the ***masmforum***

http://www.masm32.com/board/

http://www.masmforum.com/

or

SmplMath.Masm@gmx.net

Documentation for SmplMath Version 1.0.0

Document edition 1.0.0

Legal

The SmplMath macros are developed by, and for hobby programmers. There is no warranty for correctness of the macros and their produced code. The author does not assume any liability for damages (of all kinds) caused by using SmplMath macros. Furthermore, the author prohibits the usage for technologies, which can cause damage to humans or animals, willful or by technical malfunction.

The author will never earn money for SmplMath-macros. The usage of the SmplMath macros in commercial context is generally forbidden, except the Author has given his consent.

Copyright

Content

1. Abstract

While writing programs in Assembler, there are often non time critical task that requires the usage of floating point arithmetic. A good example is programming a GUI using the Windows API. In such situation it can be really exhausting to do arithmetic 'by hand', thus the usage of macros could be an improvement in point of development time, readability and reusability. The SmplMath macros follow and extents these requirements.

The key features:

- expressions as known from high level programming languages
- thread save
- supported data types: float , double, extended double, short, int, int64 (x86-87)
- global registration of symbolic constants and macro-expressions
- general purpose registers as operands (eax,ax,al,…)
- FPU registers as operands
- reuse of constants
- local and global precision control for code and constants creation
- literally equal expression are detected and only calculated once time (in the same expr.)
- extendable by user

2. Theory

How do the macros work?

Expressions are solved by first tokenizing and sorting them into an open-doubly-linked-list (ll_MathTokenize). Following this step, a second macro (ll_fSlv) will interpret this list and generates the code. This separation between parser and code-generator make the macros more flexibility in point of supported instruction sets – however, currently only the FPU is supported.

Schematic

Let's look at the follow expression: **3*x^2/2+1*2**

What we want is a (execution-) list that looks similar to this:

Table 1: execution list

Step	Operation	Description
1	**res = x^2*3/2**	this expression can be solved from left to right
2	**push res**	push intermediate result
3	**res = 1*2**	
4	**pop + res**	pop intermediate result and add it to the last result (1*2)

We get this by tokenizing the expression from left to right into a linked list, whereas each token stored in a Node

Table 2: Schematic of expression analyses (3*x^2/2+1*2)

Step	Node 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	3														
2	3	*													
3	3	*	x												
	Exponent detected: take the last operand (x) and place it before the operand in node 1. The x gets replaced by a *pop*. A *push* is placed following the ^. The next operand must be stored between ^ and the *push*.														
4	x	^	push	3	*	pop									
5	x	^	2	push	3	*	pop								
	Product (/) detected: continue operand-placement at end of list, because products has a lower priority than the exponents.														
6	x	^	2	push	3	*	pop	/							
7	x	^	2	push	3	*	pop	/	2						
	Sum (+) detected: add the operator and remember this position in list.														
8	x	^	2	push	3	*	pop	/	2	+					
9	x	^	2	push	3	*	pop	/	2	+	1				
	Product detected (*): mul has a higher priority than plus -> get the last operand and place it before the plus.														
10	x	^	2	push	3	*	pop	/	2	1	*	push	+	pop	
11	x	^	2	push	3	*	pop	/	2	1	*	2	push	+	pop

An implicit push

This must be interpreted as follow: add the last result with the one stored through an implicit push

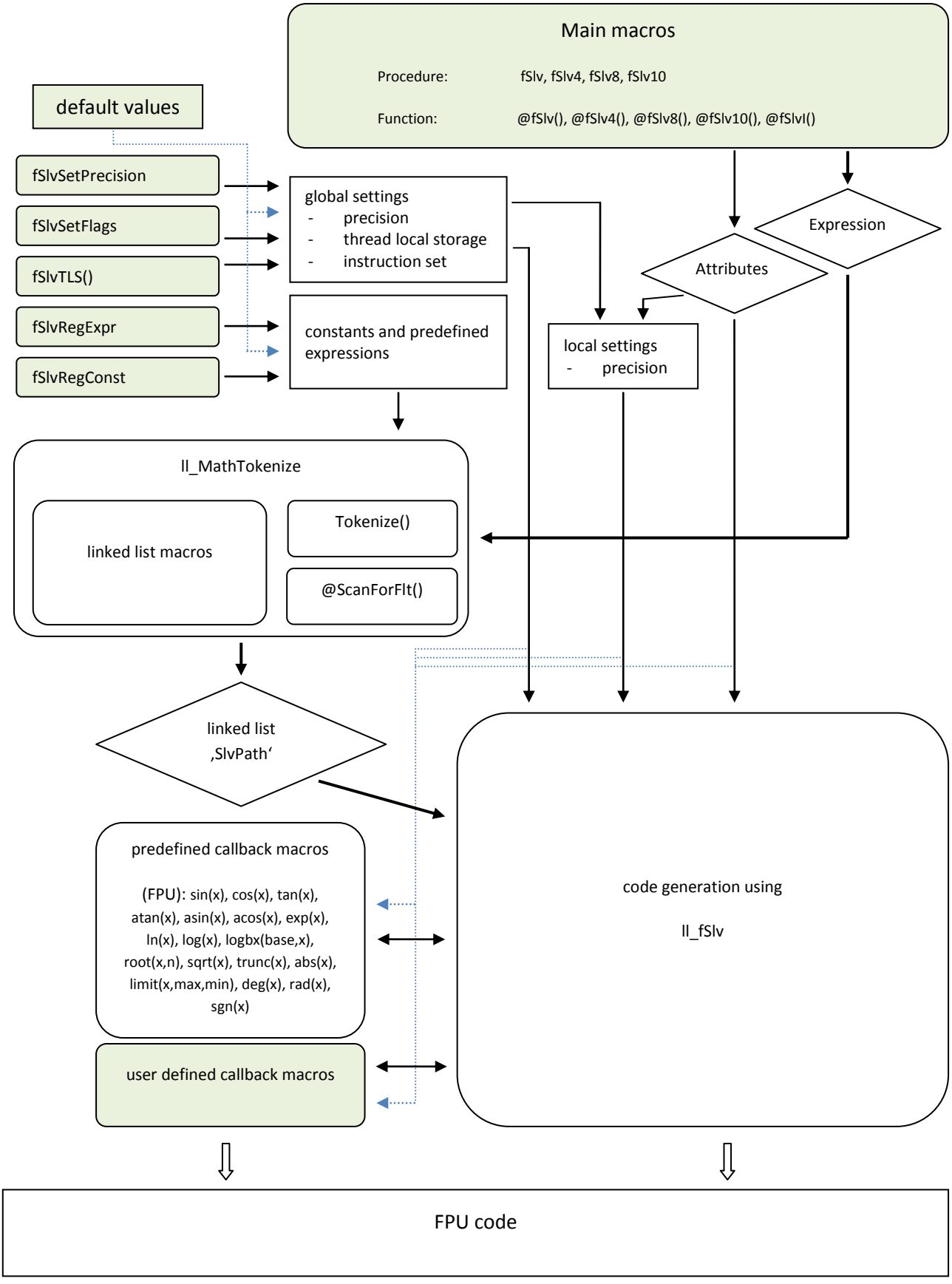
With a list as shown in Table 2, it is directly possible to create FPU code. The flowing picture shows the code created by the ll_fSlv() macros (precision = default = REAL8)

```
00401086 . D945 9C       FLD DWORD PTR SS:[EBP-64]
00401089 . DCC8          FMUL ST,ST
0040108B . DC0D 00604000 FMUL QWORD PTR DS:[fslv_real8_0]
00401091 . DC35 10604000 FDIV QWORD PTR DS:[fslv_real8_1]
00401097 . D9E8          FLD1
00401099 . DC0D 20604000 FMUL QWORD PTR DS:[fslv_real8_2]
0040109F . DEC1          FADDP ST(1),ST
```

Picture 1: created code (OllyDbg)

Block diagram

The following diagram shows the relationships between the major macros.



3. Usage

3.1 Requirements

- SmplMath macros are working with MASM (6-10) and JWASM.
- Currently only x86-32 is supported
- For assembling the examples, the MASM32 SDK is needed

3.2 Installing

Copy the folder ***macro*** to your hard drive. You can also place the includes in your project folder or any other location, but this requires to change the relative paths in \macros\SmplMath***math.inc***.

3.2.1 Setting up MASM

The SmplMath macros are case sensitive. Also ***align 16*** is used, which requires to enable SSEx-Instruction sets. Generally the flowing directives should use: ***.686p***, ***.mmx*** and ***.xmm***.

3.3 Example program

```
.686p
.model flat, stdcall
option casemap :none
.mmx
.xmm     ; needed! (align 16)

include \macros\smplmath\math.inc

.code
main proc
LOCAL x:REAL8

         fSlv x = 1 +2*3^4

         ret
main endp
end main
```

3.4 general Syntax for expression evaluation

The syntax is for both, procedure and function-like macros, the same:

fSlv [*dest*] = *expression*

@fSvl4([*dest2* =] *expression*) ; returns a REAL4-variable

The Destination operand *dest* is optional – for the procedural macros (fSlv, fSlv4,…) the result then is stored in the FPU register ST(0). The function like macros (@fSlv4(),@fSlv8(),…) allows optional to specify an additional destination *dest2* to the one returned – the return value through *EXITM* <> may be the same as *dest2*, if the type of *dest2* is equal to macros return type:

```
@fSlv4(MyReal4 = …)     →     the macro returns MyReal4
@fSlv4(MyReal8 = …)     →     the macro returns an anonym REAL4-variable
```

3.5 expressions

Expressions can consist of sums (+-), product (*/), exponents (x^y), function calls and bracket terms.

The syntax is very similar to C/C++ ones. The well known priority of operators is the same: e.g. a product has a higher priority than a sum. Currently there is only one exception for signs, which have the highest priority:

```
-x^2    is interpreted as:      (-x)^2
```

This problem is solvable by using brackets. This behavior may be correct/change in future versions of SmplMath.

General there is no nesting limit for bracket terms. However, not each expression is valid synthesizable because of limit number of available FPU registers.

3.5.1 Bracket terms

If your expression has two or more occurrences of the same bracket term, the parser will recognize this, as long as they are literally equal. This cause the bracket term to be calculated only once times:

```
x1 = (-p/2)+( (-p/2)^2 - q)^0.5    ; (-p/2) will be calculated only once time
x1 = (-p/2)+( ( -p/2)^2 - q)^0.5   ; "-p/2" <=> " -p/2" -> for the parser two different
```

Also the parser support implicit multiplication if one bracket term is directly followed by a second:

```
(x^2)(y+1)      →      (y^2)*(y+1)
(a+1)(9*b)(c+1) →      (a+1)*(9*b)*(c+1)
```

3.5.2 Function calls

A function can called by its name followed by a bracket term, holding the comma separated argument list:

```
3*sin(2*x+1)
3 * logbx(3,x)
```

Functions names are case sensitive. Currently predefined functions are:

sin(x), cos(x), tan(x), atan(x), asin(x), acos(x), exp(x), ln(x), log(x), logbx(base,x), root(x,n), sqrt(x), trunc(x), abs(x), limit(x,max,min), deg(x), rad(x), sgn(x)

There is also the possibility to add your own function – see chapter 5. Extensions.

3.5.3 Numeric values (constants)

Numeric values can be written as integer or as floating point values (MASM syntax):

Table 3: syntax for numeric values

Variation	Syntax	Examples
1	[+-]{0-9}[Tt]	-12345 , +12345t
2	[+-]{0-9}[0-9a-fA-F]{Hh}	+0ab3fh , 0FFFFFFFFh
3	[+-]{0-9}{.}[0-9]	-999.99 , 12.0 , 1.
4	[+-]{0-9}{.}[0-9]{Ee}[+-]{0-9}	+10.9E-12 , 3.E8
	{}=needed , []=optional leading blanks are ignored (space/tab)	

3.5.4 Predefining expressions

The SmplMath parser allows the usage of predefined expressions, which are called like functions. Predefined expressions behave like macros, which will be expanded while parsing.

For defining an expression, the macro fSlvRegExpr can be used:

Example: $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

```
LOCAL x:REAL4
…
; probability density function: pdf(x,μ,sigma)
fSlvRegExpr <pdf>, 3, 1/sqrt(2*pi*((arg3)^2))*exp(-(((arg1)-(arg2))^2/(2*((arg3)^2))))
…
fSlv REAL8 ptr [edx] = pdf(x,-2,0.5)
```

Generally it is a good approach to enclose all arguments ***argX*** with brackets.

3.6 Operators

The basic operators are:

+	**addition**
-	**subtraction**
*****	**multiplication**
/	**division**
^	**exponentiation**

The priority order generally decreases from left to right in an expression, except for exponents. The order between operands is (high to low): sign , () , ^ , */ , +-

3.7 Operands

3.7.1 Memory operands

Currently all FPU related memory operands are supported: SWORD, SDWORD, SQWORD, REAL4, REAL8 and REAl10.

When arithmetic for addressing is needed (SIB), it must place inside square brackets

```
myvar REAL4 2 dup (?)
…
LOCAL buffer[128]:REAL4
LOCAL var[8]:REAL4

…

fSlv SDWORD ptr [edi+4*ecx] = var[esi*4] + 1

…

fSlv SDWORD ptr [OFFSET myvar+4][8] = …
```

Square brackets are escape characters for the parser, thus any valid MASM expression can used here.

3.7.2 about constants

3.7.2.1 Predefined constants

Currently defined constants are: ***pi 1.0 1 -1 -1.0***

Using these constants (literally), cause the macros to generate corresponding FPU code. (fld1, fldpi, fchs)

Also there are some special 'known' exponent values (x^y) which cause the generation of special code instead of creating a variable/constant:

2.0 3.0 -1.0 -2.0 -3.0 0.5 -0.5

3.7.2.2 Constant values in expressions

The decision whether to generate a floating point or an integer constant depends on the notation – See chapter 3.5.3 Numeric values (constants) for more details.

However, whether to create an REAL4/8/10 or SWORD/SDWORD/SQWORD depends on the current precision settings. Each macro has its own default precision settings which can be globally overwritten by the fSlvSetPrecision macro or locally, through usage of Attributes:

Attribute	data type
i2	SWORD
i4	SDWORD
i8	SQWORD
r4	REAL4
r8	REAL8
r10	REAL10

For SWORD and SDWORD constants that are written in decimal notation the numeric

range is checked. This mechanism can also be enabled for SQWORDs by setting the global flag FSF_CHECK_SQWORD:

```
fSlvSetFlags FSF_CHECK_SQWORD
```

All constants are recorded, so that they can be reused in further macro calls – however, this requires that they are literally equal. Also, REALx constants are only reused if the types match. For Integer constants it is sufficient, if the current requested type is greater or equal to the original one.

The reuse mechanism can local disabled by the Attribute ‘no reuse’, thus new constants are created.

3.7.3 Register operands

3.7.3.1 GPRs

The macros support all general purpose registers (GPRs):

32Bit GPRs: eax, ebx, ecx, edx, esi, edi, ebp, esp

16 Bit GPRs: ax, bx, cx, dx, si, di, bp, sp

8 Bit GPRs: al, ah, bl, bh, cl, ch, dl, dh

64 Bit GPR pair: edx::eax (only usable as destination operand)

The GPRs names are case insensitive.

Loading GPRs requires to write them on the stack and then load them using FILD, thus the current set rounding mode is used for converting.

The usage of BYTE registers should be avoided.

3.7.3.2 FPU registers

The FPU register are specified by:

st0, st1, st2, st3, st4, st5, st6, st7 (case sensitive)

Using FPU registers, allows in most situations to create nearly the same code as 'by Hand'.

However, they cannot be used randomly:

- the register indexes relates to the stack before entering the macro
- keep in mind, that the macros may also need some free registers
- FPU registers can't be destination operands
- the macros detect the highest register-index in expression and assumes that all registers with lower indexes are not available for use.
- if a destination is specified, the stack will be the same as before the macro call.
- it is programmers job to clean up the register stack

Example: $x_{1/2} = -\frac{p}{2} \pm \sqrt{\left(-\frac{p}{2}\right)^2 - q}$

```
fSlv    = -p/2          ; st(0) = -p/2
fSlv    = sqrt(st0^2-q) ; st(1) = -p/2  , st(0) = sqrt((-p/2)²-q)
fSlv x1 = st1 + st0
fSlv x2 = st1 - st0     ; (better would be: fsubp st(1),st(0) , fstp x2)
fstp st(0)              ; clean up stack
fstp st(0)              ;
```

3.5.4 About using equates ([TEXT]*EQU,=*)

MASMs equates and text macros can only be used, if preceded by the expansion operator **%**. Otherwise equates are not expand, so that the SmplMath parser tread them as variables.

To bypass the need of expansion operators, the macro fSlvRegConst can be used. This macro can be seen as an extended EQU-Statement:

```
fSlvRegConst myConst,123.4
```

However, equates can still used for addressing memory operators, as long as placed between square brackets:

```
LOCAL array[10]:REAL10
…
N = SIZEOF REAL10
…
fSlv x0 = REAL10 ptr array[0*N] + x
fSlv x1 = REAL10 ptr array[1*N] + x
```

3.8 Attributes

Attributes are passed in comma separated list, which are enclosed by braces:

```
{ Attribute1, Attribute2 [,…]}
```

Each expression can have exact one list that is placed randomly in the expression. It is recommended, to place the list either at begin or at end of an expression:

```
fSlv y = x^2  {i2,r4}
```

The Attributes itself are simply strings, that must not contain braces, commas and equals sign. There is no check if an Attribute exists, thus arbitrary strings can be passed. Also they are strictly local – this means they are only valid for actually macro call. The following table shows current reserved Attributes:

Attribute	description
i2	integer precision: SWORD
i4	integer precision: SDWORD
i8	integer precision: SQWORD
r4	FP precision: REAL4
r8	FP precision: REAL8
r10	FP precision: REAL10
no reuse	create always new constants
stck:**N**	inform the macro, that **N** FPU registers are current used and not available for expression evaluation

3.9 Runtime and creation behavior

The fSlv-macros don't affect any FPU settings. Also they don't modify any GPR, except, if specified as destination operand. The stack is used for various actions: calling functions, storing and loading values, saving GPRs and so on. The flags maybe affected – for the operators ***+-*/*** this doesn't applies.

For writing thread save code it is indispensable to use fSlv-macros only inside MASM's procedures (PROC). Also this requires the usage of the fSlvTLS()-macro in each of them.

The precision control is determinate by global settings that can be overwritten locally (Attributes). These settings are never reset, which means, that the programmer is responsible for keeping track of current global precision settings (if changed).

Also it should make clear: there are two precision settings, one for floating point values/calculations, and one for integer constant/calculations. This setting may cause different code production on different precision settings.

Constants are created in the ***.const***-section. The macros assume to be called from the ***.code***-section. Constants that are used by the predefined math-functions are placed in an extra segment:

```
fpu_const SEGMENT PAGE READONLY
```

The same applies to functions, which are used for exponentiation and calculation of roots:

```
fpu_seg SEGMENT PAGE PUBLIC 'CODE'
```

3.10 Optimizations

3.10.1 The SmplMath parser and its effect on produced code

The SmplMath parser scans expressions from left to right. Whenever an operator with a higher priority than the previous one is detected, the parser places this new one before the pervious. For the calculation process this means, that the result of this operation (with higher priority) needs to be saved on the FPU stack until it is reused.

So, **generally you can optimize your expression, when writing them according to the operator's priority**. For example, this means, that it is better to place bracket term in the front, while sums should be placed at the end of an expression. However - it is clear - this not always possible.

3.10.2 Optimizing bracket terms

Bracket terms are always preferred – this means that they are calculates before any linked products, exponentiations or sums. In consequence to this, they must reserve on the FPU stack until their result is used – this can be avoided by first writing the bracket term and then the remaining expression. However, for expressions with only one argument, this doesn't apply, because the corresponding brackets are ignored by code producing macro ll_fSlv.

Literally equal bracket terms are only calculated once time. After first usage, the result is stored on the FPU stack until it is reused. For reuse, only the content of the brackets is compared – possible signs of the bracket term are treaded separately:

```
fSlv y = 1*(2+3)+3*-(2+3)       ; (2*3) is calculated only once time
```

3.10.3 Special constants

The code producing macros know some special constant values for various operations. This values cause the SmplMath macros to produce special code. For example the constant Pi(=3.141…) is known and load using corresponding FPU instruction. The same applies for the values 1 and -1, either written as integer or as floating point values (1.0,-1-0).

Furthermore the exponentiation ***x^y*** knows some special exponent values, which should be used: see chapter 3.7.2.1 Predefined constants for more details.

4. Macros

4.1 fSolve, fSlv

Type: procedure, code producing

Syntax: fSlv [dest] = expression

Attributes: i2, i4, i8, r4, r8, r10, 'no reuse', 'stck:N'

Precision: REAL8, SDWORD

Return: If the optional destination ***dest*** is not given, the result is left on the FPU stack [ST(0)]

Modifies: Flags, Stack, FPU registers

Example:

```
LOCAL x:REAL8,y:REAL8,b:DWORD
…
fSlv b = 2
fSlv y = 3*(x+sin(x)) + b
```

4.2 fSlv4/8/10

Type: procedure, code producing

Syntax: fSlv4/8/10 [dest =] expression

Attributes: i2, i4, i8, r4, r8, r10, 'no reuse', 'stck:N'

Precision: REAL4/8/10, SDWORD

Return: If the optional destination ***dest*** is not given, the result is left on the FPU stack [ST(0)]

Modifies: Flags, Stack, FPU registers

Example:

```
LOCAL x:REAL8,y:REAL8,b:DWORD
…
fSlv4 y = 123 + b
fSlv10 y = 3*(x+sin(x)) + b + 800000000   {i8}   ; force SQWORD support
```

4.3 @fSlv4/8/I

Type: function, code producing

Syntax: @fSlv4/8/I([dest =] expression)

Attributes: i2, i4, i8, r4, r8, r10, 'no reuse', 'stck:N'

Precision: REAL4/8, SDWORD

Return: Returns a variable typecasted according to the macros name. The integer version @fSlvI() returns always SDWORDs or, if specified as **dest**, 32Bit GPRs. If local storage is available, a local variable is retuned - otherwise it's a global (_BSS). If the optional destination ***dest*** is given, it will be returned if its type is the same as the macros return-type.

Modifies: Flags, Stack, FPU registers

Other notes: **Do not use** these macros with ***.ELSEIF*** or ***.WHILE*** – it won't work as expected!

Example:

```
LOCAL x1:REAL4,x2:REAL4,p:DWORD,q:DWORD,szBuffer[128]:BYTE
…
fSlv p = 3
fSlv q = 1
mov szBuffer[0],0
strcat ADDR szBuffer,"x1 = ", real4$( @fSlv4(x1 = (-p/2) + sqrt( (-p/2)^2 - q ) )
),chr$(13,10)
strcat ADDR szBuffer,"x2 = ", real4$( @fSlv4(x2 = (-p/2) - sqrt( (-p/2)^2 - q ) ) )
invoke MessageBox,0,ADDR szBuffer,0,0
…
.if @fSlvI(x1+x2) == eax
        …
.endif
```

4.4 fEQ/NE/LT/LE/GT/GE

Type: function, code producing

Syntax: fEQ/LT/LE/GT/GE(***a*** , ***b*** [, ***pByte***])

a and ***b*** are the operands to compare. This can be constants, variables and GPRs. The optional parameter ***pByte*** is a pointer to a BYTE variable, which is returned by the macro in the form <BYTE ptr ***pByte***>, after setting the byte according to the comparison result.

The default precision is REAL8 and SDWORD. This can be change by specifying an Attribute list with the parameter ***a*** or ***b***.

Attributes: i2, i4, i8, r4, r8, r10, 'no reuse'

Precision: REAL8, SDWORD

Return: Returns a Boolean BYTE-variable (0, 1). If local storage is used, this is a local variable, otherwise its global (_BSS).

Modifies: Flags, Stack, FPU registers

Other notes: **Do not use** these macros with ***.ELSEIF*** or ***.WHILE*** – it won't work as expected!

Example:

```
LOCAL x:REAL8,y:REAL8,b:DWORD
…
.if fEQ(x , y) || fGT(b , 1E-7)
    …
.endif
.if fLT(123.4 , x {r4}) ; force precision to REAL4 -> 123.4
    …
```

4.5 faEQ

Type: function, code producing

Syntax: faEQ(***a*** , ***b*** [, ***f***] [, pByte])

This function is nearly identically to the other compare macros (fEQ,fGT,..) except, that a additional tolerance factor ***f*** can be specified. The following pseudo code shows how comparison is done:

```
if a+abs(a*f) >= b && a-abs(a*f) <= b
   return 1;
else
   return 0;
```

The default value for ***f*** is 1.E-2 ≙ 1%. The data type of ***f*** can only be REAL4 or REAL8.

Other notes: read the Chapter 4.4 fEQ/NE/LT/LE/GT/GE.

4.6 fSlvTLS

Type: function, code producing

Syntax: *LOCAL* fSlvTLS([***name***][,***cb***])

This macro must place following the *LOCAL*-statement in a procedure (PROC). The purpose of fSlvTLS() is to share some local stack space with the fSlv-macros. This allows creating thread save code. Not using this macro may cause warning generation by function-like macros (e.g. @fSlv4(),…).

The first parameter ***name*** is optional and is maybe useful for debugging purpose.
cb is the number of bytes to allocate. The default value is 16.
If there are code lines in your procedure, that calls one or more function–like macros, it possible that the available storage is too small: in this case errors occur. To fix these errors, requires increasing the ***cb*** parameter. Also read the error messages carefully – they give you a hint how many bytes needed – especially the last TLS-error for the corresponding code-line.
If ***cb*** is specified, the value is round up to the next multiple of 4.

Return: Returns an expression like: *fslv_tls[16]:BYTE*

Example:

```
myProc proc param1:DWORD
LOCAL fSlvTLS()
…
myProc endp
```

4.7 fSlvRegConst

Type: procedure

Syntax: fSlvRegConst ***name***,***value***

This macro allows registering a symbolic constant. This is the same as using MASM's EQU (=) directive. However, it is not possible to use these equates, because the parser treads them as variables. Reasoned this, fSlvRegConst was introduced. See also Chapter *3.5.4 About using equates (EQU,=)*

Example:

```
fSlvRegConst e,2.718281828
fSlvRegConst three,3
```

4.8 fSlvSetPrecision

Type: procedure

Syntax: fSlvSetPrecision ***name***,[***realPrec***],[***intPrec***]

Use this macro to change the default precision of specified fSlv-macro.
The ***name*** parameter must enclose by angle brackets **<>**:
<fSlv>, <fSlv4>, <fSlv8>, <fSlv10>, <@fSlv4>, <@fSlv8>, <@fSlvI>, <fEQ>, <fNE>, …
realPrec specifies the precision for floating point constants:
REAL4, REAL8, REAL10

intPrec set the integer precision:
SWORD, SDWORD, SQWORD

Example:

```
fSlvSetPrecision <@fSlv10>,,SQWORD  ; change only integer precision
```

4.9 fSlvSetFlags/ fSlvRemoveFlags / fSlvResetFlags

Type: procedure

Syntax: fSlvSetFlags ***flags***
fSlvRemoveFlags ***flags***
fSlvResetFlags

These macros allow manipulating the global flags. The ***flags*** parameter can be a combination of the following flags:

Flag	description
FSF_REUSE_BY_TYPE	reuse integer constants only if the types are equal
FSF_CHECK_SQWORD	Check the numeric range for SQWORDs if written in decimal notation. e.g.: -12345678912

4.10 fSlvRegExpr

Type: procedure

Syntax: fSlvRegExpr ***name***, ***nargs***, ***expression***

This is a very powerful macro, which allows predefining of expressions. Using this macro, you can reduce expressions complexity and increase readability. Also this allows breaking MASMs line length limit, which is something around 260 characters.

The ***expression*** can have arguments named ***argX*** with X>=1. The number of arguments is specified by ***nargs***.
The predefined expressions are later called like function:

name(arg1,arg2[,…])

The ***expression*** is implicit enclosed by brackets. Redefinition is possible.

The usage of predefined functions is nothing more than text replacement, thus it generally a good approach to enclose all arguments ***argX*** with brackets.

Example: The following example use fSlvRegExpr to implement a Fourier series.

```
LOCAL x:REAL8
…
;/* register expression with one argument (=arg1) */
fSlvRegExpr <Tri>,1,arg1^(-2)*cos(arg1*x)+(arg1+2)^-2*cos((arg1+2)*x)+(arg1+4)^-2*cos((arg1+4)*x)

;/* fourier series: triangular pulse */
fSlv REAL8 ptr [edx] =  Tri(1)+Tri(7)+Tri(13) +Tri(19)+Tri(25)\
                        +Tri(31)+Tri(37)+Tri(43)+Tri(49)+Tri(55)\
                        +Tri(61)+Tri(67)+Tri(73)+Tri(79)+Tri(85)\
                        +Tri(91)+Tri(97)+Tri(103)+Tri(109)
```

4.11 fpuSetPrecision

Type: procedure, code producing, FPU helper macro

Syntax: fpuSetPrecision [**mem16**],[**prec**]

With this macro, the FPU's precision settings can be changed.

mem16 is an optional, WORD-sized memory location that is used for reading and writing FPU's control word. The default value is: <WORD ptr [esp-2]>

The wished precision can be indicated by the parameter ***prec***:
REAL4, REAL8, REAL10
The default value is REAL8

Modifies: Stack, Flags, FPU control word

Example:

```
fpuSetPrecision ,REAL4
```

4.12 ldl

Type: procedure, code producing, FPU helper macro

Syntax: ldl ***var*** = ***value*** [,…]

Load local REAL4/8, [S]DWORD or [S]QWORD variables with a constants.

The macro uses the stack for copying (push imm32). Zero-values (<0>,<0.0>) are detected and cause a MOV-construct, instead of push/pop.

Modifies: Stack

Example:

```
LOCAL x:REAL8,y:REAL4,a:SDWORD,b:SQWORD
…
ldl x=3.5, y=1, a=5, b = 1316496516568
```

4.13 r4/r8IsValid

Type: function, code producing, FPU helper macro

Syntax: r4/r8IsValid(***mem***)

Test if variable ***mem*** is a valid floating point value (REAL4/8).

Returns: The macros return an expression that can be evaluated by using MASM's HLL constructs : <!ZERO?>

Modifies: Stack, Flags

Example:

```
.if r4IsValid(myreal4)
    ; Valid
.elseif
    ; Invalid
.endif
```

4.14 fSlvStatistics

Type: procedure, output to build console

Syntax: fSlvStatistics

Shows some statistic in the build console: number of macro calls, created and reused constants,…

5. Extensions

5.1 Requirements

For adding own functions it required to be familiar with:
- MASM's macro system
- FPU
- x86-32 calling conventions
- thread safety

The following table gives a quick overview about SmplMath-files:

file name	description
SmplMath.inc	contains macros for end-user
math_tokenizer.inc	the parser
code_generator.inc	code producing macros
linked_list.inc	
math_functions.inc	predefined functions.
misc.inc	miscellaneous macros

5.2 Concept

A function must not add explicit to the macros. The parser *ll_MathTokenize* detects function calls by syntax and assumes that the corresponding function exists. The code producing macro *ll_fSlv*, that use the parsers result, then check for a so called 'function descriptor', which is a macro-function, that returns basic information about the used function. If this descriptor is found, the corresponding code producing macro is called (function).
So, for adding a function, two macros must be declared:

- the function descriptor
- the code producing macro (function)

Both macros must take care for the current precision settings. This information is shared through global equates:

equate	**values**
fslv_lcl_real_type	4=REAL4, 8=REAL8, 10=REAL10
fslv_lcl_int_type	2=SWORD, 4=SDWORD, 8=SQWORD

This equates must not changed – only read them.

It is recommended to create different code according to current precision settings.
For example, if the precision of REAL10 is requested, corresponding sized constants must load explicit using FLD, whereas REAL4 or REAL8 constants could be specified as memory operands. So, a REAL10 version would need one more additional FPU register.

Also there is the option to access Attributes using the macro @fslv_test_attribute(). Using these macro you can process own defined Attributers, if wished or needed.

Currently arguments of a function call are passed through the FPU stack by pushing them from left to right. The return value is left on the stack (st(0)).

5.3 function descriptor

The minimum declaration for function descriptor:

```
fslv_fnc_&FncName&_dscptr macro flags:req
        IF flags AND FNCD_VERSION
                EXITM <1>               ; compatible with version 1
        ELSEIF flags AND FNCD_NARGS
                EXITM <&nArgs>          ; number of arguments
        ELSEIF flags AND FNCD_NREGS
                EXITM <&nRegs>          ; number of registers used additional to the ones used for augment passing
        ELSE
                EXITM <0>
        ENDIF
endm
```

The descriptor must process these three messages and return values according to the current precision settings. There are two helper macros, which simplify the declaration of descriptors: default_fnc_dscptr and type_depending_fnc_dscptr.

If a function uses SSEx, the descriptor must process FNCD_USES_SSE and return <1>.

5.4 function (code producing macro)

The prototype for a function:

```
fslv_fnc_&FncName& macro
…
endm
```

The arguments are pushed on the FPU stack from left to right. These must be removed before leaving the macro. The return value must stored in ST(0). It can be assumed, that the macro is called from the ***.code***-section.

Some consideration about written code:

- do not use global variables – use the stack instead.
- do not change the FPU settings
- do not modify any GPR
- use different sized constants according to current precision settings
- do not modify FPU registers below your arguments
- do not use anonym labels **@@** - instead use the macro-LOCAL-directive for creating labels. Alternating you can use the macro get_unique_lbl().
- do not use SEGMENT/ENDS. Nesting segments sometimes cause MASM to get weird ideas.
- do not call any of the code producing macros (e.g. fSlv) – this kind of nesting is not possible.

If your function really needs global variables, you should use Windows thread local storage. Also you can use Attributes to pass a global location.

5.5 Example

The following example shows the declaration of the function ***rad()***, which converts a radian value to degree. This is done by multiplying the argument with (180/pi).

```
.const
    align 16
    fpu_const_r4_r2d    REAL4  57.295779513082320876798154814105
    align 16
    fpu_const_r8_r2d    REAL8  57.295779513082320876798154814105
    align 16
    fpu_const_r10_r2d   REAL10 57.295779513082320876798154814105
...

fslv_fnc_deg macro
    IF fslv_lcl_real_type LE 8
        IF fslv_lcl_real_type EQ 4
            fmul fpu_const_r4_r2d
        ELSE
            fmul fpu_const_r8_r2d
        ENDIF
    ELSE
        fld fpu_const_r10_r2d               ; load constant
        fmulp
    ENDIF
endm

fslv_fnc_deg_dscptr macro flags:req
    IF flags AND FNCD_VERSION
        EXITM <1>
    ELSEIF flags AND FNCD_NARGS
        EXITM <1>                           ; the function has one argument
    ELSEIF flags AND FNCD_NREGS
        IF fslv_lcl_real_type LE 8          ; REAL4/8 versions doesn't need extra registers
            EXITM <0>
        ELSEIF fslv_lcl_real_type EQ 10
            EXITM <1>                       ; the REAL10-verion uses one additional register
        ENDIF
    ELSE
        EXITM <0>
    ENDIF
endm
```

The following shows the original declaration, which can be found in math_functions.inc:

```
fslv_fnc_deg macro
        IF fslv_lcl_real_type LE 8
                fmul @CatStr(<fpu_const_r>,%fslv_lcl_real_type,<_r2d>)
        ELSE
                fld @CatStr(<fpu_const_r>,%fslv_lcl_real_type,<_r2d>)
                fmulp
        ENDIF
endm
type_depending_fnc_dscptr <deg>,1,0,0,1
```

5.6 helper macros

The following section documents macros, which are useful for extending SmplMath macros. Some of them are made for general purpose.

Most macros return information through equates and text macros – this make them very powerful.

The equates and text macros names are always prefixed with a shorten form of the macros name: e.g. @ScanForFlt() returns equates and text macros with the suffix ***sff_*** ➔ ***S**can**F**or**F**lt.*

5.6.1 default_fnc_dscptr

Location: \SmplMath\ math_functions.inc

Type: procedure, helper macro, part of SmplMath-System

Syntax: default_fnc_dscptr macro ***FncName***:req, ***nArgs***:=<1>, ***nRegs***:=<0>

This maco creates a default function descriptor.
nArgs is the number of function arguments and ***nRegs*** the number of register additional used to the ones used for argument passing.
The function name ***FncName*** should be enclosed by angle brackets.

Example: From math_functions.inc:

```
default_fnc_dscptr <cos>,1
default_fnc_dscptr <tan>,1,2
```

5.6.2 type_depending_fnc_dscptr

Location: \SmplMath\ math_functions.inc

Type: procedure, helper macro, part of SmplMath-System

Syntax: type_depending_fnc_dscptr macro
FncName:req, ***nArgs***:=<1>, ***nRegs_r4***:=<0>, ***nRegs_r8***:=<0>, ***nRegs_r10***:=<0>

Creates a function descriptor and allows declaring different register usage according to current REAL-precision settings.

Example: From math_functions.inc:

```
type_depending_fnc_dscptr <rad>,1,0,0,1
```

The function ***rad***, need on more register for REAL10, because a constant is loaded using FLD.

5.6.3 @fslv_test_attribute

Location: \SmplMath\ code_generator.inc

Type: function, part of SmplMath-System

Syntax: @fslv_test_attribute macro ***name***:req, ***separator***, ***default_value***

This macro can be used to check and get the specified attribute ***name***.
Optionally the function can parse attributes in the form:

<***name***><**separator**><***value***>

e.g.: *stck: 3*
separator must be one character.
The extracted string ***value*** is returned through the text macro *fsta_value*. If the attribute is not found, *fsta_value* is filled with ***default_value***.

[Access to the raw data, the attribute list, is also possible: first check that the equate *fslv_attributes?* is set to is 1 – this indicates that a attribute list is available. Then you can use the list, which stored in the text macro *fslv_current_attributes*]

Return: Returns one, if the attribute exist, otherwise it returns zero.
The following equates and text macros are returned:

name	**type**	**description**
fsta_found	equation (=)	equal to EXITM <> {0,1}
fsta_value	TEXTEQU	If ***separator*** is used, it returns either the extracted string or, if the attribute is not found, ***default_value***

5.6.4 @GetArgByIndex

Location: \SmplMath\misc.inc

Type: function, general purpose macro, string macro, dependences: none

Syntax: @GetArgByIndex macro ***index***:req, ***default***:=<>,***args***:VARARG

Extract an argument form the comma separated list ***args*** by its zero based ***index***. If **index** is out of range, ***default*** is returned.

Returns: extracted argument or specified default value.
This argument is also supplied through the text macro *gabi_arg*.

5.6.5 @MatchStrI

Location: \SmplMath\misc.inc

Type: function, general purpose macro, string macro, dependences: macro: @TrimStr()

Syntax: @MatchStrI macro ***txt***:=<>,***trim***:=<0>,***args***:VARARG

Makes a case insensitive compare of ***txt*** with all arguments ***args***[i].

If ***trim*** is 1, all leading and trailing spaces/tabs are removed before comparing. If leading and trailing spaces/tabs are wished to be part of comparison, ***txt*** and the corresponding argument in ***args***, must doubly enclosed by angle brackets << … >>.

Return: one based index of matching argument in ***args***, or zero if no match occurs. Also returned:

name	type	description
msi_index	equate,=	the same as EXITM <>
msi_txt	TEXTEQU	= ***txt*** If ***trim*** is set, msi_txt contains the trimmed string.

5.6.6 @ScanForFlt

Location: \SmplMath\misc.inc

Type: function, general purpose macro, dependences: none

Syntax: @ScanForFlt macro ***pos***:req, ***txt***:=<>, ***Chr***

Scans in given string ***txt***, beginning from position ***pos***, for a floating point or integer value. The integer value can be written in hexadecimal or decimal notation. There is no size-checking, only syntax (masm-specific):

Variation	Syntax	Examples
1	[+-]{0-9}[Tt]	-12345 , +12345t
2	[+-]{0-9}[0-9a-fA-F]{Hh}	+0ab3fh , 0FFFFFFFFh
3	[+-]{0-9}{.}[0-9]	-999.99 , 12.0 , 1.
4	[+-]{0-9}{.}[0-9]{Ee}[+-]{0-9}	+10.9E-12 , 3.E8
	{}=needed , []=optional	
	leading blanks are ignored (space/tab)	

Chr is an optional string, containing all characters that can follow a numeric expression. If you specific "default" then for the follow char. is tested

space, tab, +, -, *, /,) , ","

Another key word is "blank", which cause to test for spaces and tabs.
The ends of strings always match. If this parameter is blank, the macro doesn't take care of characters following a valid numeric expression, even if these invalidate it.

The scanner is greedy - that means, it will ‘eat’ as much char. as possible.

Returns: If no numeric value found, zero is returned.

For floating point values 1 is returned, for integer values 2. Whether the integer is written in decimal or hex decimal notation, can be determinate by testing the equate *sff_flag* with SFF_HEX_SUFFIX-flag.

If ***pos*** is out of range, zero is returned and *sff_type* is set to -1.

Returned text macros and equates:

name	**type**	**description**
sff_type	equate, =	same as EXITM <>
sff_flag	equate,=	combination of flags, see next table for more details
sff_numstr	TEXTEQU	extracted numeric expression (all blanks are removed)
sff_pos	equate, =	points to next char. following numeric string (this could be out of string-range)
sff_num_pos	equate, =	points to begin of number in ***txt***

The following table shows all flags for *sff_flag*:

valid for			**Flag**	**Description**
float	int	int(hex)		
X	X	X	SFF_SIGN	the number has an sign
X			SFF_EXPONENT	an exponent is present
X			SFF_EXP_SIGN	the exponent has a sign
	X		SFF_DECIMAL_SUFFIX	the decimal-number ends with an "t" or "T"
		X	SFF_HEX_SUFFIX	the integer is written in hexadecimal notation

The following text macros depend on the flags and/or the current numeric type:

name	**flag**	**type**	**description**	**characters**
sff_sign	SFF_SIGN	1,2	sign of num	<+->
sff_pre_decimal		1,2	pre-decimal positions	<0123456789>
sff_fract_digits		1	post decimal positions	<0123456789>
sff_exp_sign	SFF_EXP_SIGN +SFF_EXPONENT	1	sign of the exponent	<+->
sff_exp_digits	SFF_EXPONENT	1	exponent-digits	<0123456789>
sff_hex_digits	SFF_HEX_SUFFIX	2	hex-digits	<0123456789abcdefABCDEF>

Examples:

```
Exampel:
  1       @ScanForFlt(1,< 1.0E-19 >)     --> 1 , sff_numstr=<1.0E-19> , sff_pos=9
  2       @ScanForFlt(1,<-1.0*x>)        --> 1 , sff_numstr=<-1.0>    , sff_pos=5
  3       @ScanForFlt(1,< -100 + x>)     --> 2 , sff_numstr=<-100>    , sff_pos=6
  4       @ScanForFlt(1,<-1E-2>)         --> 2 , sff_numstr=<-1>      , sff_pos=3
  5       @ScanForFlt(1,< 0ffh >)        --> 2 , sff_numstr=<0ffh>    , sff_pos=6

  6       @ScanForFlt(1,<12()>)          --> 2 , sff_numstr=<12>      , sff_pos=3
  7       @ScanForFlt(1,<12()>,default)  --> 0
  8       myChars TEXTEQU <   +-*/()>
          @ScanForFlt(1,<12()>,myChars)  --> 2 , sff_numstr=<12>      , sff_pos=3
```

5.6.7 @IsRealType/ @IsIntegerType

Location: \SmplMath\misc.inc

Type: function, general purpose macro, dependences: none

Syntax: @IsRealType macro memOpattr:req
@IsIntegerType macro memOpattr:req

This two macros test, whether the given memory operand is a real or integer type. Also they return the operands size in bytes:
4=REAL4, 8=REAL8, 10=REAL10
2=[S]WORD, 4=[S]DWORD, 8=[S]QWORD

Return: returns the types size in bytes or zero, if the types does not match.
This return value is mirrored in the equate *irt_type* for @IsRealType() and *iit_type* for @IsIntegerType().

5.6.8 get_unique_lbl

Location: \SmplMath\misc.inc

Type: function, general purpose macro, dependences: equate: *unique_num_cntr*

Syntax: get_unique_lbl macro ***name***:req

Creates a unique string by concatenating ***name*** with a consecutive number.

Example:

```
get_unique_lbl(<some_name_>)  => some_name_0
get_unique_lbl(<some_name_>)  => some_name_1
get_unique_lbl(<some_name_>)  => some_name_2
…
```

5.6.9 @TrimStr

Location: \SmplMath\misc.inc

Type: function, general purpose macro, string macro, dependences: none

Syntax: @TrimStr macro txt:=<>

Removes all leading and trailing spaces/tabs

Return: trimmed string. This string is mirrored in the text macro *ts_txt*.

5.6.10 @RepAllStr

Location: \SmplMath\misc.inc

Type: function, general purpose macro, string macro, dependences: none

Syntax: @RepAllStr macro ***pos***:req, ***txt1***:=<>,***txt2***:=<>,***txt3***:=<>

Replace all occurrences of ***txt2*** in ***txt1*** with ***txt3***. The operation is case sensitive and starts from position ***pos***.

Return: processed string. This result is mirrored in the text macro *rpa_txt1*.

5.6.11 @ToLowerCase

Location: \SmplMath\misc.inc

Type: function, general purpose macro, string macro, dependences: none

Syntax: @ToLowerCase macro ***txt***:=<>

Converts string ***txt*** to lower case.

Return: converted string. This result is mirrored in the text macro *tlc_txt*.

5.6.12 @RemoveAllStrI

Location: \SmplMath\misc.inc

Type: function, general purpose macro, string macro, dependences: macro: @TrimStr()

Syntax: @RemoveAllStrI macro ***txt***:=<>, ***args***:VARARG

Remove all occurrences of ***args***[i] in ***txt***. The operation is case insensitive.
Working with blanks requires the usage of double angle bracket: << >>

Return: modified string.
The following text macros are also returned:

name	**description**
rasi_out	EXITM <>
rasi_txt	modified string in lower case
rasi_lst	Comma separated list of indexes (zero based) in ***args***. Each index equals an occurrences ***args***[i] in ***txt***

5.7 about MASM's bugs

While looking through the SmplMath macros, you will find various places, whit comments like "bugfix for masm". The following list describes some of MASM bugs:

- MASM often cooks, when nesting segments. This cause strange code and data creation/mixing.
- The following construct produce sometimes errors:

```
IF @some_macro() …
…
ELSEIF @some_macro2() …
…
ELSE
…
ENDIF
```

  This bug can be solved by splitting the construct:

```
IF @some_macro() …
ELSE
    IF @some_macro2() …
        …
    ELSE
        …
    ENDIF
ENDIF
```

- Passing expressions with deep bracket nesting let MASM sometimes fail.
-