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

http://sourceforge.net/projects/smplmath/

https://github.com/qwordAtGitHub/SmplMath

SmplMath.Masm@gmx.net

with updates by **HSE**.

https://github.com/ASMHSE/SmplMath

Documentation for SmplMath Version 2.1.0

Document edition 1.1.0

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 the SmplMath macros system.

Copyright

The SmplMath macros-system can be copied and shared by everyone, as long as it contains all original, unmodified files (documentation, include files and examples).

Individual macros can be extracted and reused in other projects, as long as they do not break with the author's legal statements. Also the following comment lines must add immediately before the macro declaration (this must done for each macro):

```
;/*-----------------------------------------------------*/
;/* The macro MacroName                                 */
;/* is part of the SmplMath macros system.              */
;/* copyright by qWord @ www.masmforum.com , 2011-2017 */
;/* SmplMath.Masm{at}gmx{dot}net                        */
;/* http://sourceforge.net/projects/smplmath/           */
;/*-----------------------------------------------------*/
```

MacroName must be replaced by the corresponding macro name.

Example:

```
;/*-----------------------------------------------------*/
;/* The macro fSlW                                      */
;/* is part of the SmplMath macros system.              */
;/* copyright by HSE @ www.masmforum.com , 2019-2022    */
;/* https://github.com/ASMHSE/SmplMath/                 */
;/*-----------------------------------------------------*/
```

The same applies to functions/procedures that are contained in the libraries. In this case it the word "macro" must be replaced by the word "function" or "procedure".

Example:

```
;/*-----------------------------------------------------*/
;/* The function fpu_st1_pow_st0                        */
;/* is part of the SmplMath macros system.              */
;/* copyright by qWord @ www.masmforum.com , 2011-2017 */
;/* SmplMath.Masm{at}gmx{dot}net                        */
;/* http://sourceforge.net/projects/smplmath/           */
;/*-----------------------------------------------------*/
```

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
- Instruction sets*: x86-32/64, FPU, SSE2
- thread save
- supported data types*: float , double, extended double, short, int, int64 (x86-87)
- symbolic constants and macro-expressions
- General purpose registers as operands (rax,eax,ax,al,…)
- FPU and XMM registers as operands*
- reuse of constants
- constant expressions may be solved while assembling
- local and global precision control for code and constants creation
- equal expressions are detected and only once time calculated (in the same macro call)
- front end / back end design
- Extendable by user: functions and back ends can be added

** Depends on the current selected back end*

2 Overview

The SmplMath macros can be roughly categorized in three classes:

- **Expression evaluation:**

  create code for given mathematically expressions or compare values

- **Environment management:**

  control global settings like volatile registers, used instruction set/ back end, precision settings, ...

- **Helper macros:**

  code or data producing macros: local storage, initializing local variables, changing the FPU's precision, creating constants, ...

The SmplMath macros implement an front end / back end architecture. The front end is the parser that converts the mathematically expression into an token list respecting the priority of operators. The back ends use this list and then create corresponding code.

Currently there are three back ends:

- **FPU back end**: creates only FPU code
- **SSE2 back end**: uses SSE2 as far as possible, but may also use the FPU
- **Integer back end**[1]: use x86/x64 general purpose instructions

All back ends take care of the equate @WordSize, thus they can create code for x32 and x64 programs.

3 Theory

3.1 How do the macros work?

Expressions are solved by first tokenizing and sorting them into an open-doubly-linked-list, whereas the priority of operators is considered (ll_MathTokenize). Following this step, a second macro (ll_fSlv) will interpret this list and call macros from the current selected back end, which then create the code. It should be noted that term "back end" is not fully correct, because these macros primarily create code, whereas the interpretation and most of the optimization is done in ll_fSlv.

[1] This back end does not creates very efficient code and should be avoided

3.2 Schematic

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

What we want is an 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

implicit push

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

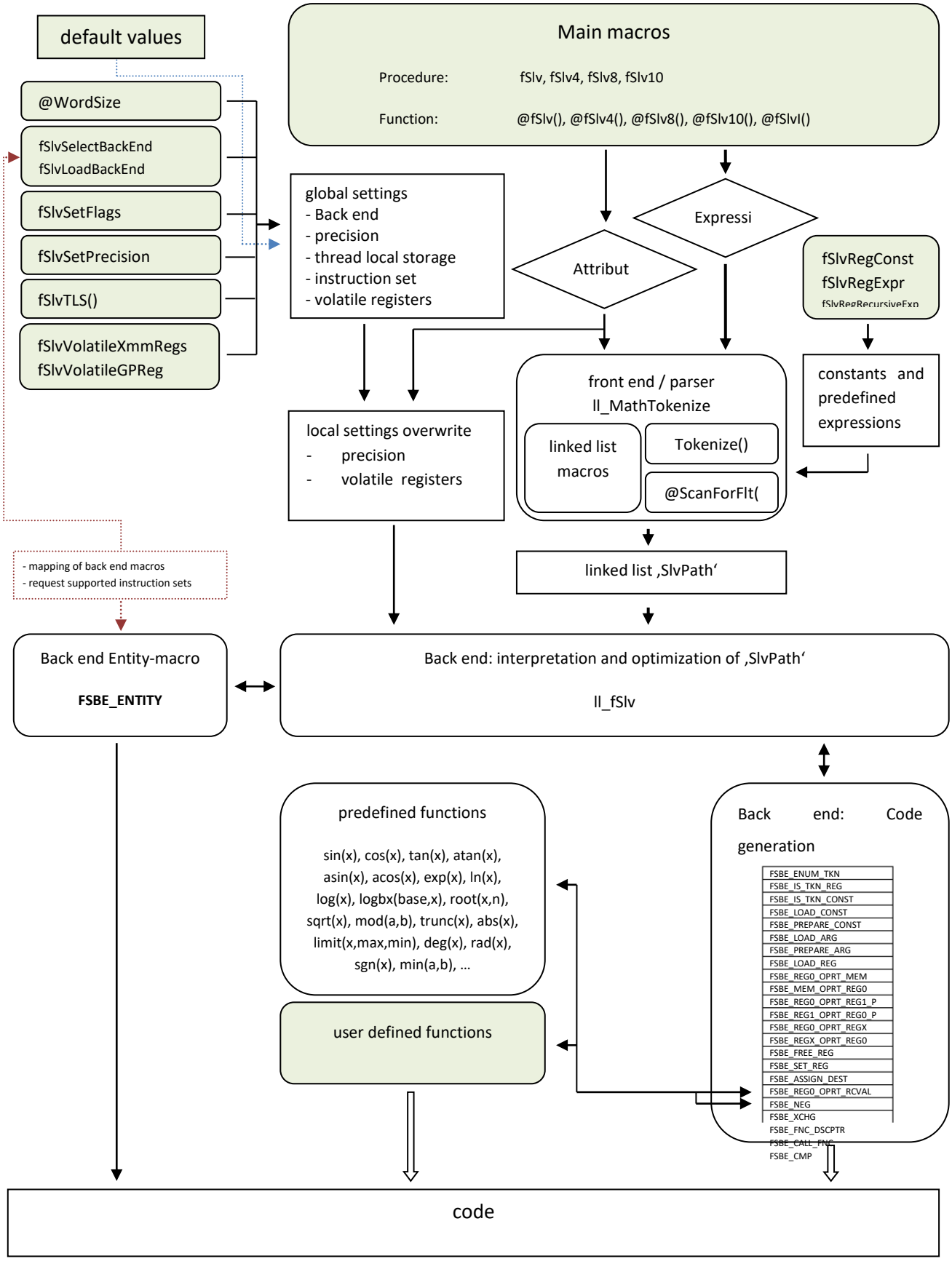
With a list as shown in Table 2, it is directly possible to create FPU code. The flowing picture shows the code created by the FPU back end (precision = default = REAL8)

```
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)

3.3 Block diagram

The following diagram shows the relationships between the major macros.



4 Usage

4.1 Requirements

- the SmplMath macros are working with MASM (6-11) and JWASM[1].
- for x86-64, JWASM is needed. ML64 supported.
- the MASM32 SDK[2] and the WinInc[3] include files are needed for assembling the examples

4.2 Installing

Copy the folder ***macros*** to the root directory of your hard drive.

(other locations requires to change the relative paths in *math.inc*, *macros.inc* and for all examples)

4.2.1 Setting up MASM and JWASM (x86-32)

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

4.2.2 Setting up JWASM (x86-64)

The macros assume that the stack is aligned to 16 and that the shadow space is allocated. The example 4.3.2 shows one way to accomplish these requirements.

4.2.3 Setting up the SmplMath environment

4.2.3.1 *Choosing the Instruction set / back end*

The default back end gets loaded according to target architecture. For 32-bit programs the FPU back end is used, for x64 the SSE2/FPU back end. Currently there are three macros that allows to switch the back end:

fSlvSelectBackEnd

fSlvSetFlags / fSlvRemoveFlags

fSlvLoadBackEnd

The following code shows solutions that will work for both, x32 and x64 programs:

```
        fSlvSelectBackEnd FPU,SSE2  ; load the SSE2/FPU back end
        fSlvSelectBackEnd FPU       ; load the FPU back end
        fSlvSelectBackEnd           ; load the integer back end (not recommended)
```

[1] http://sourceforge.net/projects/jwasm/ or http://www.japheth.de/
[2] http://www.masm32.com
[3] http://sourceforge.net/projects/wininc/ or http://www.japheth.de/

4.3 Example programs

4.3.1 x86-32 Example

```
.686
.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
```

4.3.2 x86-64 Example

```
option casemap:none
option frame:auto

JWASM_STORE_REGISTER_ARGUMENTS  EQU 1
JWASM_STACK_SPACE_RESERVATION   EQU 2

option win64:JWASM_STACK_SPACE_RESERVATION  ; shadow space is always allocated in PROCs

include \macros\SmplMath\math.inc

.code
main proc FRAME
LOCAL x:REAL8

        fSlv x = 3^4*5

        ret

main endp
end main
```

4.4 syntax for expression evaluation

The syntax is for procedural and function-like macros is the same:

fSlv *[dest]** = *expression [, [[dest$_n$] =] expression$_n$,...]*

@fSvl4(*[dest2]* = *expression [, [[dest$_n$] =] expression$_n$,...]*) ; returns a REAL4-variable

The destination operand *dest* is optional for the FPU back end. In this case the result is stored in ST(0). The function like macros (@fSlv4(),@fSlv8(),...) allows 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
```

The macros can solve several comma separated expression, which are solved from left to right in the list. Omitting *dest$_n$* let the corresponding result stay on the register stack.

* The exact syntax depends on the current selected Back end. For example, the SSE2 and the Integer back end requires an destination operand for the procedural macros.

4.5 Expressions

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

The priority of operators is the same as for many HLLs: 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.

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

4.5.1 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  expressions
```

Also the parser supports 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)
```

4.5.2 Function calls

A function can be 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:

Table 3: available functions

name	x64	FPU	SSE2	Data types[1]	Remarks
sin(x)	X	X		R4/8/10	
cos(x)	X	X		R4/8/10	
tan(x)	X	X		R4/8/10	
atan(x)	X	X		R4/8/10	
atan2(y,x)	X	X		R4/8/10	atan(y/x)
asin(x)	X	X		R4/8/10	
acos(x)	X	X		R4/8/10	
exp(x)	X	X		R4/8/10	e^x ; e = 2.71828...
ln(x)	X	X		R4/8/10	base = e = 2.71828...
log(x)	X	X		R4/8/10	base = 10
logbx(base,x)	X	X		R4/8/10	
root(x,n)	X	X		R4/8/10	n= +-(1,2,3,...)
root2(x,a,b)	X	X		R4/8/10	a and b = ±1,±2,±3,... ; result = $x^{(a/b)}$
sqrt(x)	X	X	X	R4/8/10	
trunc(x)	X	X		R4/8/10	
mod(a,b)	X	X		R4/8/10	
abs(x)	X	X	X	R4/8/10	
limit(x,max,min)	X	X	X	R4/8/10	
deg(x)	X	X	X	R4/8/10	radian to degree
rad(x)	X	X	X	R4/8/10	degree to radian
sgn(x)	X	X	X	R4/8/10	returns sign of x: -1 or +1
min(a,b)	X	X	X	R4/8/10	
max(a,b)	X	X	X	R4/8/10	
rsqrtss	X		X	R4	error less equal $1.5*2^{-12}$
rcpss	X		X	R4	error less equal $1.5*2^{-12}$

SSE2 back end: for functions that are not implemented in SSE2, the FPU version is used.

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

[1] REAL10 is only supported by the FPU back end

4.5.3 Numeric values (constants)

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

Tabelle 4: 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
5	[+-][0]{0-9a-fA-F}{Rr}	3F800000r, 03F800000r
	{}=needed , []=optional	
	leading blanks are ignored (space/tab)	

The hexadecimal notation of floating point values (5.) is restricted by the current precision settings. MASM expected 8 digits for REAL4, 16 for REAL8 and 20 for REAL10. One more leading zero is allowed and required if the numbers begins with A-F or has leading zeros.

4.5.4 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. The whole function term is implicit enclosed by brackets.

Recursive expression are also possible using fSlvRegRecursiveExpr.

4.6 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 , () , ^ , */ , +-

The exponentiation is currently calculated by using the FPU. For negative bases, the function checks if the exponent is an integer value. In this case the result's singe is changed, if the exponent is odd. The result for negative bases in combination with a non-integer exponents is always positive. Also the function response to some special values:

0^0=1 , 0^x=0 , 0^{-x}=div. by zero.

4.7 Operands

4.7.1 Memory operands

Currently all FPU related memory operands are supported: SWORD, SDWORD, SQWORD, REAL4, REAL8 and REAl10. However, this applies only to the FPU back end. The SSE2 back end does not support SQWORD and REAL10.

When arithmetic for addressing is needed (SIB, scale index base), 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 be used here.

4.7.2 about constants

4.7.2.1 *Predefined constants*

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

Using these constants (literally), cause the FPU back end to generate corresponding FPU code. (fld1, fldpi, fchs). The SSE2 back end use corresponding memory operands.

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

4.7.2.2 *Constant values in expressions*

The decision whether to generate a floating point or an integer constant depends on the used back end and 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, which 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, REAL4/8/10 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.

The SSE2 back end force creation of floating points constants in any case. This means that integer expression are transformed to FP values considering the precision settings: for REAL4 it is check whether the value can be exact represented. For REAL8 there is currently no checking mechanism.

4.7.2.3 *constants calculations*

The SmplMath macros may solve integer expressions while assembling (all basic operators). Also, divisions by constants may be replaced by a multiplication with the reciprocal.

[It is also planned to add support for calculations of floating point constants.]

4.7.3 Register operands (GPRs)

The macros support all general purpose registers (GPRs):

64-bit GPRs:	rax,rbx,rcx,rdx,rsi,rdi,rbp,rsp,r8,r9,r10,r11,r12,r13,r14,r15
32-bit GPRs:	eax,ebx,ecx,edx,edi,esi,ebp,esp,r8d,r9d,r10d,r11d,r12d,r13d,r14d,r15d
16-bit GPRs:	ax,bx,cx,dx,di,si,bp,sp,r8w,r9w,r10w,r11w,r12w,r13w,r14w,r15w
8-bit GPRs:	al,ah,cl,ch,dl,dh,bl,bh,sil,dil,bpl,spl,r8b,r9b,r10b,r11b,r12b,r13b,r14b,r15b
64-bit GPR pair:	edx::eax (only usable as destination operand)

The GPR names are case insensitive. All registers are assumed to be signed (SBYTE, SWORD, SDWORD and SQWORD)

Loading GPRs requires writing them on the stack and then load them using FILD. If used as destination operator, the current rounding mode is used for conversion (FISTP for the FPU, cvtss/d2si for SSE2).

4.7.4 FPU registers

If the FPU back end is used, the register can be specified by:

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

Using FPU registers allows in many situations to create nearly the same code as 'by Hand'. However, it should be proved if the problem can be solved by using multiple expression in one call to an evaluation macro.

FPU registers can't 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 only be destination operands for the procedural macros fSlv and fSlv4/8/10. Also only st0-st6 can be used.
- the macros detect the highest register-index in expression and assumes that all registers with lower indexes are not available for use (this can be overwritten by the stck-Attr.).
- use the 'stck'-Attribute, to inform the macro, how many registers are currently used and not available for the macro.
- 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
- use the 'pop' and the 'free' Attribute to clear the 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) {stck:1}        ; st(1) = -p/2  , st(0) = sqrt((-p/2)²-q)
fSlv x1 = st1 + st0     {stck:2}        ;
fSlv x2 = st1 - st0     {stck:2,pop:2}  ; pop -> clean up stack
```

The same example using multiple assignments in one macro call:

```
fSlv x1 = (-p/2)+(sqrt((-p/2)^2-q)) , \     ; line continuation using '\'
     x2 = (-p/2)+(sqrt((-p/2)^2-q))
```

Please take care of the comma and the backslash in the first line.

The following picture shows the generated code for compare (register usage (left), multiple expressions (right)):

```
FLD QWORD PTR SS:[LOCAL.6]
FCHS
FMUL QWORD PTR DS:[402000]
FLD ST
FMUL ST,ST
FSUB QWORD PTR SS:[LOCAL.8]
FSQRT
FLD ST(1)
FADD ST,ST(1)
FSTP QWORD PTR SS:[LOCAL.2]
FLD ST(1)
FSUB ST,ST(1)
FSTP QWORD PTR SS:[LOCAL.4]
FSTP ST
FSTP ST
```

```
FLD QWORD PTR SS:[LOCAL.6]
FCHS
FMUL QWORD PTR DS:[402000]
FST ST(1)
FLD ST
FMUL ST,ST
FSUB QWORD PTR SS:[LOCAL.8]
FSQRT
FST ST(3)
FADDP ST(1),ST
FSTP QWORD PTR SS:[LOCAL.2]
FLD ST
FSUB ST,ST(2)
FSTP QWORD PTR SS:[LOCAL.4]
FFREE ST
FFREE ST(1)
```

Picture 2: FPU registers usage vs. multiple expressions

4.7.5 Using XMM registers

If the SSE2 back end is used, XMM register can be specified as operands:

x32: *xmm0 - xmm7*
x64: *xmm0 - xmm15*

The names are not case sensitive. Used XMM register are treated as non-volatile, even if they are marked as volatile (see also fSlvVolatileXmmRegs).
Whether a XMM register contains a scalar double or single precision value, depends on the precision of the used macro. For example, the fSlv8 macro assumes that the registers contain REAL8 values. The fSlv4 macro needs REAL4 values.
However, XMM registers can be typecast:

REAL4 ptr xmmX
REAL8 ptr xmmX

e.g.:

```
fSlv4 xmm1 = REAL8 ptr xmm2 * 2.3
```

If the typecast differs from the macro precision setting, the register content is converted. In the above example the scalar double value in XMM2 is converted (CVTSD2SI) to a scalar single value before doing the multiplication.

Integer values in XMM registers are not supported.

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

MASM's 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
```

4.8 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 equal signs. 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 [*ST(0),ST(1),...,ST(**N**-1)*] are currently used and not available for expression evaluation.
pop:**N**	pop the FPU stack N-times after calculation
free:***i***	free the register *st*(***i***) after calculation
free: ***i1*-*i2***	free the registers *st*(***i1***…***i2***) after calculation

4.9 Runtime and creation behavior

The fSlv-macros don't affect any FPU and XMM 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.

It is assumed that the FPU stack is empty when calling fSlv-macros. If the FPU stack is not empty , the 'stck'-Attribute must be used to inform the macros about that.

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:

SmplMthC SEGMENT READONLY

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

SmplMath SEGMENT PUBLIC 'CODE'

Pleas remarks that these segment names has changed for SmplMath version 2+[1]

Some SSE2 function that are created dynamically, will be placed in the following segment:

IRegFncs SEGMENT READONLY PUBLIC 'CODE'

For x64, the macros assume that they can use the register shadow space (Microsoft FASTCALL) and that the stack is aligned to 16.

The XMM registers follows the Windows ABI / calling convention: for x32 all XMM registers are assumed to be volatile, for x64 only xmm0-xmm5. These registers can be changed using fSlvVolatileXmmRegs.

[1] Previous the names was: < fpu_const SEGMENT PAGE READONLY > and < fpu_seg SEGMENT PAGE PUBLIC 'CODE' >

4.10 Optimizations

4.10.1 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 register 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.

4.10.2 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 reister 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 the code producing macro ll_fSlv.

Literally equal bracket terms are only calculated once time. After first usage, the result is stored on the register 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
```

4.10.3 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.

4.10.4 recommendations for the SSE2 back end

- Variables should have the same precision as the evaluation macro to avoid unnecessary conversions (CVTxx2yy instructions). For example, use the fSlv4 macros for REAL4 variables.
- Make at least EAX (or better EAX and EDX) volatile by using fSlvVolatileGPReg.
  This applies especially when using GPRs or integer variables as arguments, because the macros use EAX (and EDX) for conversion and must thus save them when they are not volatile.

4.11 Modular programming

Since version 2, the SmplMath-System has a static library that allows modular programming. There is no need to include this library explicit, because it is done in the include file *math.inc*.

The __MODUEL__ constant, which was used in previous versions, is no longer used and should be removed from existing code.

5 Macros

5.1 fSolve, fSlv

Type: procedure, code producing

Syntax: fSlv [dest] 1 = expression [,[dest2 =] expression2,...]

Attributes[1]: i2, i4, i8, r4, r8, r10, 'no reuse', 'stck:N', 'free:i', 'free:i1-i2', 'pop:N'

Precision: REAL8, SDWORD

Return: The destination ***dest*** is optional for the FPU back end. In that case the result is stored in ST(0). The SSE2 (and Integer) back end requires an destination ***dest***. If multiple expressions are solved, the corresponding destinations are optional for all back ends. However, for the SSE2 (and Integer) back end the results destinations are undefined and thus lost. For the FPU the unassigned results are pushed on the FPU stack (left to right).

Modifies: Flags, Stack, FPU/XMM registers1

Example:

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

5.2 fSlv4/8/10

Type: procedure, code producing

Syntax: fSlv4/8/10 [dest =] 1 expression [,dest2 = expression2,...]

Attributes[1]: i2, i4, i8, r4, r8, r10, 'no reuse', 'stck:N'

Precision: REAL4/8/10, SDWORD

Return: see fSlv

Modifies: Flags, Stack, FPU/XMM registers1

Example:

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

[1] depends on the selected back end

5.3 @fSlv4/8/I/I64

Type: function, code producing

Syntax: @fSlv4/8/I/I64[1]([dest =] expression)

Attributes[2]: i2, i4, i8, r4, r8, r10, 'no reuse', 'stck:N'

Precision: REAL4/8, SDWORD,SQWORD1

Return: Returns a variable typecasted according to the macros name.

The integer versions @fSlvI() and @fSlvI64() returns always SDWORD/SQWORDs or, if specified as **dest**, 32/64-bit GPRs. Also they use REAL8 precision.

If local storage is available, a local variable is returned - 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 macro return-type.

Modifies: Flags, Stack, FPU/XMM registers1

Remarks: @fSlvI64() is only available for x64 programs.

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

Example:

```
;/* strcat and real4$ are string macros of the MASM32 SDK */
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
```

[1] only available for x64 programs
[2] depends on the selected back end

5.4 fEQ/NE/LT/LE/GT/GE

Type: function, code producing

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

a and ***b*** are the operands to compare. This can be constants, variables and GPRs.

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

Attributes[1]: i2, i4, i8, r4, r8, r10, 'no reuse'

Precision: REAL8, SDWORD

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

Modifies: Flags, Stack, FPU/XMM registers1

Other notes: **Do not use** these macros with ***.ELSEIF*** or ***.WHILE*** – it won't work as expected! Early expansion also is a problem if more that one comparison is used with **.IF**. Instead you can use **Conditional flow macros**

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
    …
.endif
```

5.5 faEQ

Type: function, code producing

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

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 abs(a-b) <= abs(a*f)
   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 fEQ/NE/LT/LE/GT/GE.

[1] depends on the selected back end

5.6 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 4**@WordSize*.
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 *@WordSize*.

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

Example:

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

5.7 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 About using equates (EQU,=)

Example:

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

5.8 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
```

5.9 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_USE_FPU	Load the FPU back end. This is only done if FSF_USE_SSE2 is not set.
FSF_USE_SSE2	Load an SSE2 back end. Currently the FSF_USE_FPU is automatically added. Future versions of SmplMath may load an SSE2-only back end. If this is not wished, add the FSF_USE_FPU explicit.
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
FSF_NO_INT_CONSTS	Integer constants are converted to real constants. This flag is ignored, if the used back end can't handle real constants.

The current flags are returned by the macro function fSlvGetFlags().

5.10 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.

[1]If the expression for an argument ***argX*** begins with '***#***', it is treated as an integer expression that can be solved by MASM's preprocessor (for an example, see fSlvRegRecursiveExpr)

For declaring recursive expression see fSlvRegRecursiveExpr.

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)
```

Other notes: See also fSlvRegRecursiveExpr

[1] This paragraph refers on calling a predefined function. For clarification see the example of fSlvRegRecursiveExpr

5.11 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/rsp-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
```

5.12 ldl

Type: procedure, code producing, helper macro

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

Load local REAL4/8, [S]DWORD or [S]QWORD variables with a constants.
[TEXT]EQUs can also used as ***value***.
Integer values are transformed to FP values by adding the suffix <.0>.

x32: the macro may uses the stack for copying (push imm32).
x64: the macro may use *push/pop rax + mov rax,imm64*

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
```

5.13 r4/r8IsValid

Type: function, code producing, 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
```

5.14 fSlvStatistics

Type: procedure, output to build console

Syntax: fSlvStatistics

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

5.15 _R4/8/10

Type: function, data producing, helper macro

Syntax: _R4/8/10(***value***) or R4/8/10(***value***)[only x86-32]

Creates a REAL4/8/10 constant in the *.const*-section. The ***value*** can be an integer or floating point initialize (hexadecimal-notation is not supported). These macros reuse constants, if they are literally equal.

SmplMath 2+: Because R8 and R10 are reserved word for x64 (=registers), the macros has renamed by adding an underscore as prefix. However, for 32-bit programs the old names are still available and the new variations are added as EQUs.

Return: A REAL4/8/10 constant.

Example:

```
        invoke function,…,_R4(45),_R4(12.0),…
```

5.16 fTest

Type: function, code producing

Syntax: fTest(***a*** , ***b*** [, $\Delta_{\mathbf{Digits}}$])

This function determine, if the value ***a*** is negligible in compare to ***b***.
This is done by comparing the binary exponent values: if the difference is greater than $\Delta_{\mathbf{Digits}}$, then the function returns true:

```
if ( exponent(b) - exponent(a) <= Δ_Digits )
    return 1;
else
    return 0;
```

The default value for $\Delta_{\mathbf{Digits}}$ is 20 $\widehat{=}$ $\log(2^{20}) \approx 6$ decimal digits.

Do not compare to zero (***b***==0) – it won't work in any case.

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

Example:

```
.if fTest(1.0E-9,1.0E-2)
    ;1.0E-9 is negligible in compare to 1.0E-2
.endif
```

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

5.17 fSlvSelectBackEnd

Type: procedure

Syntax: fSlvSelectBackEnd [***token1***][, ***token2***, ...]

This macro selects and loads a back end that supports specified instruction sets. The macro compares all registered back ends and choose the one that supports at least all of the instruction sets.
The following table shows the current known tokens (case insensitive):

token	notes
x86	this token is implicit added; ignored for 64Bit programs
x64	this token is implicit added; ignored for 32Bit programs
FPU	
SSE2	

Usage of unknown tokens produce corresponding errors. If the selected back end use more instruction sets as specified, warnings are emitted.
If no token is specified, the integer back end is loaded (not recommended!).

Example:

```
fSlvSelectBackEnd x86,FPU,SSE2
```

Other notes: see also fSlvLoadBackEnd and fSlvSetFlags/ fSlvRemoveFlags / fSlvResetFlags.

5.18 fSlvLoadBackEnd

Type: procedure

Syntax: fSlvLoadBackEnd ***name***

Loads the specified back end by its ***name***.

name	description
<x86-32/64,FPU>	Uses only the FPU
<x86-32/64,SSE2,FPU>	Uses SSE2 and the FPU if needed
<x86-32/64>	Use general purpose integer instructions

The angle brackets are needed because of the commas in the names.

Example:

```
fSlvLoadBackEnd <x86-32/64,SSE2,FPU>
```

Other notes: see also fSlvSelectBackEnd and fSlvSetFlags/ fSlvRemoveFlags / fSlvResetFlags.

5.19 fSlvRegRecursiveExpr

Type: procedure

Syntax: fSlvRegRecursiveExpr ***name***, ***nargs***, ***rest***, ***expression***

This macro is an enhanced version of fSlvRegExpr, which allows to create recursive predefined expressions.

The ***expression*** can have arguments named ***argX*** with X>=1. The number of arguments is specified by ***nargs*** and must be at least one. The first argument must be an integer expression that can be solved by MASM's preprocessor. If this argument reach zero, recursion stops. In this case, ***rest*** is used for replacement instead of ***expression***. The ***rest*** can also contain all arguments ***argX*** and must be enclosed in angle brackets <> if it contains commas.

Predefined expressions are later called like function:

name(arg1,arg2[,…])

The ***expression*** and the ***rest*** are implicit enclosed by brackets. Redefinition is possible.

[1]If an expression for ***argX*** (or the ***rest***) begins with '***#***', it is treated as an integer expression that is solved by MASM's preprocessor.

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

Please note the remarks section following the example!

Example: The following example use fSlvRegRecursiveExpr and fSlvRegExpr to implement an continued fraction for *tan(x)*:

$$\tan(x) = \cfrac{x}{1-\cfrac{x^2}{3-\cfrac{x^2}{5-\cfrac{x^2}{7-\ddots}}}}$$

[1] This paragraph refers on calling a predefined function. e.g. _ftan (#3-1,...)

```
LOCAL x:REAL8
…
;/* declare the recursive part                                          */
:/* arg1 = recursion counter, arg2 = {1,3,5,7,....} , arg3 becomes x^2 */
fSlvRegRecursiveExpr _ftan,3,<#arg2-1>,-arg3/_ftan(#arg1-1,#arg2+2,arg3)+arg2

;/**
; * wrap the recursive expression. The nesting deep is 10. arg1,arg2 and the rest
; * are integer expressions that can be solved while assembling.
; */
fSlvRegExpr ftan,1,arg1/_ftan(10,1,(arg1^2))
…
;/* use it */
fSlv x = ftan(5)
```

Remarks: The above example -expression would allow arbitrary nesting deep. This is possible because all growing arguments are integer expressions that are solved while replacing. If your expression contains arguments that grows and can't resolved while assembling, the nesting deep is limit by MASM's line length -> at some point MASM will throw errors (in tmt_get_expr...).

Another limitation for the nesting deep are the available registers. In the example arg2 is added to the negated fraction term. A naive way would be to write: *arg2-arg3/_ftan(...)*

But this configuration force the solver to store arg2 on the register stack and then solve the following expression - the back end will quickly get out of registers and throw corresponding errors.

Other notes: See also fSlvRegExpr

5.20 fSlvVolatileGPReg

Type: procedure

Syntax: fSlvVolatileGPReg ***operation, reg1,reg2,...***

This macro manipulates the global list of volatile general purpose registers (GPR). By default all registers are treated as nonvolatile.
The argument ***operation*** is an literal that specify what to do with the list:

operation	description
add	Add the registers to list; duplicates are ignored
remove	Remove the registers, if found in the list
set	Replace the current list with the specified one
default	Restore the default values; not further arguments are allowed
push	Push all current volatile registers on a virtual stack
pop	Pop volatile register from a virtual stack

The registers are specified by their name: rax, eax, al, bl, ...
However, partial registers are expanded to the full sized register.
For example AL will become RAX for x64 programs.
The push and pop operations are convergent to MASM's push/popcontext directive.

All arguments are case insensitive.

It is recommended to make R/EAX volatile, because it will increase the code quality.

Example:

```
fSlvVolatileGPReg add,eax,edx   ; eax and ecx are volatile registers
```

Other notes: See also fSlvVolatileXmmRegs

5.21 fSlvVolatileXmmRegs

Type: procedure

Syntax: fSlvVolatileXmmRegs ***operation, xmmreg1, xmmreg2,...***

This macro manipulates the global list of volatile XMM registers. For x86-32, all XMM registers are assumed to be volatile, for x64 only XMM0-XMM5 (corresponding to Microsoft's FASTCALL-calling convention)

The argument ***operation*** is an literal that specify what to do with the list:

operation	description
add	Add the registers to list; duplicates are ignored
remove	Remove the registers, if found in the list
set	Replace the current list with the specified one
default	Restore the default values; not further arguments are allowed
push	Push all current volatile registers on a virtual stack
pop	Pop volatile register from a virtual stack

The registers are specified by their name: XMM0...XMM7/15.
The push and pop operations are convergent to MASM's push/popcontext directive.

All arguments are case insensitive.

Example:

```
fSlvVolatileXmmRegs remove,xmm0,xmm1,xmm2,xmm3  ; xmm0-3 are nonvolatile
```

Other notes: See also fSlvVolatileGPReg

6 Extensions

6.1 Requirements

For adding own functions or Back ends it is required to be familiar with:

- MASM's macro system
- FPU, SSE2
- x86-32/64 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	
linked_list.inc	
math_functions.inc	predefined functions.
misc.inc	miscellaneous macros
backends.inc	
backends\x86_32_64_fpu.inc	code producing macros (Back end)
backends\x86_32_64_fpu_sse2.inc	code producing macros (Back end)
expressions.inc	some predefined expressions

6.2 adding functions

6.2.1 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 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)

FPU and SSE2 code requires two different function macros, whereas the descriptor macro is shared.

All 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.

There is the option to access Attributes using the macro @fslv_test_attribute(). With this macro, you can process own defined Attributes, if wished or needed.

The macros must also take care for the global equate fslv_glb_instrset, which describes the current used instruction sets.

Flag	description
FSIS_FPU	The FPU is used. Parameters are passed through the stack
FSIS_SSE2	SSE2 instructions are used; The function must support FNCD_IREG.
FSIS_SSE3	currently uncommitted
FSIS_SSSE3	currently uncommitted
FSIS_SSE41	currently uncommitted
FSIS_SSE42	currently uncommitted
FSIS_SSE4A	currently uncommitted
FSIS_AVX	currently uncommitted
FSIS_CVT16	currently uncommitted
FSIS_XOP	currently uncommitted
FSIS_FMA3	currently uncommitted
FSIS_FMA4	currently uncommitted

6.2.2 function descriptor

It is highly recommended to not implement the function descriptor by self. Instead use the helper macro default_fnc_dscptr2.

The minimum declaration for a function descriptor:

```
fslv_fnc_&FncName&_dscptr macro flags:req, param1:=<0>
        IF flags AND FNCD_VERSION
                EXITM <2>               ; compatible with version 2
        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
        ELSEIF flags AND FNCD_ISTCK     ; parameters are passed through the register stack
                EXITM <-1>
        ELSEIF flags AND FNCD_INSTR_SET ; supported instruction sets: must be combination of FSIS_* flags
                EXITM <FSIS_FPU>
        ELSEIF flags AND FNCD_TST_LCL_TYPES ; check if the function supports current precision settings
                EXITM <-1>
        ELSE
                EXITM <0>
        ENDIF
endm
```

The descriptor must process these messages and return values according to the current precision settings.

FNCD_VERSION	A version number. This value is currently not used, but should be set to *2*.
FNCD_NARGS	The number of arguments that the function has.
FNCD_NREGS	number of additional used registers. param1 = combination of the FSIS_*-flags
FNCD_X32	x86-32 supported
FNCD_X64	x86-64 supported
FNCD_ISTCK	function implements the *fslv_fnc_&FncName&* macro
FNCD_IREG	function implements the *fslv_fnc_ireg_&FncName&* macro
FNCD_INSTR_SET	Returns a combination of the *FSIS_** -flags that indicates the used instruction sets (SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4.A, AVX, CVT16, FMA3, XOP)
FNCD_TST_LCL_TYPES	Check if the function supports the types described by *fslv_lcl_real_type* and *fslv_lcl_int_type*. Returns none zero, if function has a implementation for the local type settings.
FNCD_IREG_RECORD_SIG	Requires FNCD_IREG. The function call is placed in the code segment ***IRegFncs*** and an RET instruction is appended. The signature of the call (= combination of the argument registers and free registers, the instruction set and the current precision settings) is recorded. If a further call match that signature, the corresponding label is called instead of the function's macro.

6.2.3 function (code producing macro)

The prototype for a function that implements FNCD_ISTCK (commonly FPU functions):

```
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.

The prototype for a function that implements FNCD_IREG (commonly SSEx functions):

```
fslv_fnc_ireg_&FncName& macro [arg1,[arg2,...]] [free1,[free2,...]] free_regs:VARARG
…
endm
```

The arguments are passed from left to right as arg1, arg2 [,...] . Also, all free volatile registers are passed through the parameter free_regs . Only free registers that has been requested by FNCD_NREGS are guaranteed to be passed to the macro - other registers must be check (IFNB <&freeN>).
The result is returned through the first argument.

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. Alternatively you can use the macro get_unique_lbl().
- avoid 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.
- x32 and x64 code can be mixed in many situations by using the @WordSize symbol, which returns 4 for x32 and 8 for x64. Also the T_REG() and T_EXPR() macros allows to create generic expressions.
- x64: by definition of the SmplMath macros, the stack BYTE ptr [RSP+0...31] (=32 byte) can be used for any purpose. Furthermore it is aligned by 16.

If your function really needs global variables, you should use Window's thread local storage. Also you can use Attributes to pass a global locations.

PROCs and constants can be placed in the following, already existing segments:

```
SmplMthC SEGMENT READONLY
    ;...
SmplMthC ENDS

SmplMath SEGMENT PUBLIC 'CODE'
    ;...
SmplMath ENDS
```

Definitions in these segments should be done in an include file and not inside the function-macro (or descriptor).

6.2.4 Example

The following example shows the declaration of the function ***deg***, which converts an radian value to degree by multiplying the value with a constant factor.

```
;/* IStck , FPU */
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

;/* IReg, SSE2 */
fslv_fnc_ireg_deg macro arg0,free_regs:VARARG
        IF fslv_lcl_real_type EQ 4
                mulss arg0,fpu_const_r4_r2d
        ELSE
                mulsd arg0,fpu_const_r8_r2d
        ENDIF
endm

;/* The function descriptor for both macros */
default_fnc_dscptr2 <deg>,nArgs=1,fpu=-1,x64=-1,SSE2=-1,nRegs_r10=1,IReg=-1
```

6.2.5 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.*

6.2.5.1 *default_fnc_dscptr2*

Location: \SmplMath\ math_functions.inc

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

Syntax: default_fnc_dscptr2 macro ***FncName***:req, ***attributes1=value1 [,attribute2=value2,...]***

This macro creates a default function descriptor. The function name ***FncName*** should be enclosed by angle brackets. The properties of the function are specified by the following tokens:

token	syntax	default value	description
version	version=value	2	the version number; should be 2.
nArgs	nArgs=value	0	number of function arguments.
nRegs	nRegs=value	0	number of registers that are used additional to the parameter registers.
IReg	IReg=Boolean value	0	implements the *fslv_fnc_ireg_&FncName&* macro.
IStck	IStck=Boolean value	-1	implements the *fslv_fnc_&FncName&* macro.
x32	x32= Boolean value	-1	has a x32 implementation.
x64	x64= Boolean value	0	has a x64 implementation / take care of @WordSize.
InstrSet	InstrSet=value	0	determines the supported instruction sets. Is a combination of *FSIS_** flags.
sse2	sse2=Boolean value	0	has a SSE2 implementation. requires IReg.
fpu	fpu=Boolean value	0	has a FPU implementation. requires IStck.
nRegs_r4	nRegs_r4=value	0	as nRegs; specific for local precision = REAL4
nRegs_r8	nRegs_r8=value	0	as above, for REAL8
nRegs_r10	nRegs_r10=value	0	as above, for REAL10
nXmmRegs_r4	nXmmRegs_r4=value	0	as for nRegs; specific for local precision = REAL4 and SSEx instruction sets.
nXmmRegs_r8	nXmmRegs_r8=value	0	as above, forREAL8
realTypes	<realTypes=<{4,8,10},...>> Whereas 4=REAL4,8=REAL8,...	<all types >	Comma separated list of accepted precision settings. e.g. *realTypes=<4,8>*. The default is that the function accept all possible precision settings.
intTypes	as above	<all types>	As above, for integer types.
Signature	Signature=Boolean value	0	enables call-by-signature for IReg.

Example: From math_functions.inc:

```
default_fnc_dscptr2 <abs>,nArgs=1,fpu=-1,x64=-1,SSE2=-1,IReg=-1
default_fnc_dscptr2 <expd>,nArgs=1,nRegs=3,x64=-1,SSE2=-1,<realTypes=<8>>,IReg=-1,Signature=-1
default_fnc_dscptr2 <rad>,nArgs=1,fpu=-1,x64=-1,SSE2=-1,nRegs_r10=1,IReg=-1
```

6.2.5.2 *default_fnc_dscptr (obsolete)*

This macro is obsolete and should not longer used. Use default_fnc_dscptr2 instead.

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 macro 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.

6.2.5.3 *type_depending_fnc_dscptr (obsolete)*

This macro is obsolete and should not longer used. Use default_fnc_dscptr2 instead.

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.

6.2.5.4 *@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 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*

6.2.5.5 *@GetArgByIndex*

Location: \SmplMath\misc.inc

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

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

Extracts 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*.

6.2.5.6 *@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.

6.2.5.7 *@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
5	[+-][0]{0-9a-fA-F}{Rr}	3F800000r , 03F800000r
	{}=needed , []=optional	
	leading blanks are ignored (space/tab)	

Chr is an optional string, containing all characters that can follow a numeric expression. This test applies to the character directly following the numeric expression (one char. is tested). If you specific "default" then for the following char. is tested

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

Another key word is "blank", which cause to test for space and tab.
The key word "blank**s**" let the macro test **all** remaining characters in the string. 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	float (hex)	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
	X			SFF_HEX_REAL	floating point value in hexadecimal notation
	X			SFF_HEX_REAL4/8/10	the type of FP value when 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
```

6.2.5.8 *@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().

6.2.5.9 *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
```

6.2.5.10 *@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*.

6.2.5.11 *@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*.

6.2.5.12 *@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*.

6.2.5.13 *@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***

6.2.5.14 *T_REG*

Location: \SmplMath\backends.inc

Type: function, general purpose macro, for generic code, dependences: none

Syntax: T_REG macro ***reg32***:req

Returns the 64 bit counterpart of the specified 32 bit registe for x64 applications. For x32, the input register is returned.

reg32 = {eax,ebx,ecx,edx,esp,edi,esi,ebp}

The register names are not case sensitive.

Return: 32 or 64 bit register according to @WordSize.

Example:

```
fld REAL8 ptr [T_REG(edx)]
```

6.2.5.15 *T_EXPR*

Location: \SmplMath\backends.inc

Type: function, general purpose macro, for generic code, dependences: none

Syntax: T_EXPR macro ***x86_32_expr***,***x86_64_expr***

Returns ***x86_32_expr*** for x32 applications and ***x86_64_expr*** for x64 applications.

Return: ***x86_32_expr*** or ***x86_64_expr*** according to @WordSize.

Example:

```
movzx eax,WORD ptr T_EXPR([esp-2],[rsp])
```

6.2.6 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.
- GOTO over several IF/ELSE/ENDIF constructs can cause that MASM lost track of nesting. Typical errors are something like "... cannot have more than one ELSE clause per IF block". This problem can be solved by trying to reformulate/remove ELSE clauses in the corresponding range or (better) discard the GOTOs.
- MASM v8-11 crashes, if a macro returns a floating point initializer written in hexadecimal notation that is larger than 8 digits. But this seems to only occurs if the macro is used to initialize a variable:

```
.data
        foo REAL8 macroFnc()  ; returns e.g. 3fe0000000000000r
.code
; solution:
foo2 TEXTEQU macroFnc()
foo REAL8 foo2
```

7 Complementary macros

Some new and old macros are included for easy and better programing.

7.1 Loop macros

These are based in Chapter 8 of The Art of Assembly Language by Randall Hyde

7.1.1 Integer Loops

By default counter variable must be DWORD size in 32 bits or QWORD size in 64bits, and auxiliar register used is EAX/RAX. Other way you can define an auxiliar register (and then signed integer size).

7.1.1.1 *Ascending Integer Loop*

This is Hyde's macro. Use: ForLp, ExitFor, CutFor and Next.

ForLp to begin loop and **Next** to close loop are requiered.

Optionaly , **CutFor** finish the step, and **ExitFor** finish the loop.

Example:

```
    local v1:dword                                          ; 32 bits

    ForLp v1, 0, 5
        .if eax == 3
            CutFor v1
        .endif
        print str$(v1), 13, 10

    Next v1
```

7.1.1.2 *Modified Ascending Integer Loop*

Use: ForLp_df, ExitFor, CutFor and Next.

ForLp_df begin loop decrementing number of steps. This is used for arrays, because in arrays first position is "**0"(zero)** and last position is **"size-1"**

Example:

```
        local v1:qword, v1_t:qword, sizeofarray: qword          ; 64 bits

        mov sizeofarray, 4
        ForLp_df v1, 0, sizeofarray, r13
            .if r13 == 1
                 CutFor v1
            .endif
            conout str$(v1), lf
        Next v1
```

7.1.1.3 *Descending Integer Loop*

Use: ForLpN, ExitFor, CutForN and NextN.

Example:

```
local v2:word                                                   ; 64 bits

 ForLpN v2, 5, 0, cx
     .if cx == 3
         CutForN v2
     .endif
     xor r11  , r11
     mov r11w , v2
     conout str$(r11), lf
 NextN v2
```

7.1.2 Floating Point Ascending Loop

Use: ForLpR and NextR.

Counter variable, begin, end and step size must be REAL4/8 variables.

Example:

```
        local v3: real4                                          ; 32 bits

        ForLpR v3, FP4(0.0), FP4(2.0), FP4(0.5) ; counter, begin, end, step
            print real4$(v3), 13, 10
        NextR
```

Note: ExitFor can work, but in this kind of problems have almost not purpose. Also is easy to make a cut but never was needed.

7.2 Compound operators macros

7.2.1 fSlvW

The fSlW macro is wrapper to fSlv that allow compound operators: *=, /=, +=, -=, ++ and --

Example:

```
        fSlvW res1 = 10.0 + 5.0

        fSlvW res1 ++

        fSlvW res1 -= -10

        fSlvW res1 *=2

        mov eax, 0
        fSlvW res1[eax*8] *= 2
```

7.2.2 fSlvB

The macro allow compound operators: &=, |=, ^=, {{ and }} for operations AND, OR, XOR, SHL and SHR.

By default auxiliar GPR is DWORD size in 32 bits or QWORD size in 64bits: EAX/R10. Other way you can define an auxiliar register (and then size).

Examples:

```
item dd 0
res1 dd 0
res2 dw 0
BIT equ 10

fSlvB eax ^= BIT

fSlvB res2 &= BIT, dx



fSlvB res1 |= item



fSlvB BIT &= 5

fSlvB eax {{ BIT

fSlvB res1 {{ BIT



fSlvB res1 }} ítem
```

```
xor eax  ,  BIT

mov dx, res2
and dx ,  BIT
mov res2 , dx

mov eax, res1
or eax , item
mov res1 , eax

error

shl eax  ,  BIT

mov eax, res1
shl eax ,  BIT
mov res1 , eax

mov eax, res1
mov ecx,  ítem
shr eax , cl
mov res1 , eax
```

7.3 Pseudo push/pop macros in 64 bits

This is a little macro system to simulate pushs/pops. It's adapted from SmplMath results storing system. Work in 32/64 and can be thread safe/unsafe. Size of registers can be 1, 2, 4 or 8 bytes.

To be thread safe **fregTLS()** must be declared as local, and is similar to fSlvTLS.

The intended purpose of this macros it's for easy transformation of 32 bits code to 64 bits code, and for dual bitness programming.

7.3.1 Pseudo push/pop registers

freg_push / **freg_pop**: store and retrive GPR from stack.

freg_peek retrieve in a GPR last value stored without remotion.

Example:

```
 local fregTLS()

  conout "    rax", tab
  mov rax, 1350
  freg_push rax
  mov rax, 2264
  freg_pop rax
  conout str$(rax),lf
```

7.3.2 Pseudo push/pop variables

freg_pushv / **freg_popv**: store and retrive variables from stack.

freg_peekv retrieve in a variable last value stored without remotion.

These requiere a GPR to move value (by default are EAX and R10 but you can use other).

Example:

```
freg_pushv [xax].SDLL_ITEM.pNextItem, R11
....
freg_pop xax
```

7.3.3 Corrections

freg_correction is a not so automatic correction for unbalanced number of push/pop. That happen in conditional flow. Must be **positive** before extra **pop** and **negative** before extra **push**.

Example:

```
 freg_push xax
.if [xsi].BibBigMaster.options.TextEdition
      invoke CheckMenuItem, xax, IDM_TEXT_ED, MF_UNCHECKED
      freg_pop xax
      invoke CheckMenuItem, xax, IDM_BLOCK_ED, MF_CHECKED
.else
      invoke CheckMenuItem, xax, IDM_TEXT_ED, MF_CHECKED
      freg_correction +1
      freg_pop xax
      invoke CheckMenuItem, xax, IDM_BLOCK_ED, MF_UNCHECKED
.endif
```

7.3.4 Pseudo pushad/popad

freg_pushad / **freg_popad**: store and retrive all x86 registers (or extensions) to/from stack. That is for example EAX in 32 bits and RAX in 64 bits.

Example:

```
freg_pushad
....
freg_popad
```

7.4 Conditional flow macros

These macros are based in Mabdelouahab's macros, with symbols modifications following Steve Hutchesson sugerences. Extension of macros allow floating point numbers comparisons for conditional flow easily. Usually this it's not so simple because early macro expansión by most assemblers, and can be a serius problem using HLL conditional flow.

Problems arise using **.IF** with more that one comparison or simple comparison with ***.ELSEIF*** or ***.WHILE.***

Macros names are using symbol "@" to prevent colisions with masm64 SDK when using ML64, and internal HLL reserved words in JWAsm family.

Comparisons:

equality tests:
- **eq** equal
- **ne** not equal

signed comparisons:
- **gt** signed greater than
- **lt** signed less than
- **ge** signed greater than or equal
- **le** signed less than or equal

unsigned comparisons:
- **ua** unsigned above
- **ub** unsigned below
- **ae** unsigned above or equal
- **be** unsigned below or equal

floating point comparisons:
- **fEQ** equal
- **fNE** not equal

- **fGT** greater than
- **fLT** less than
- **fGE** greater than or equal
- **fLE** signed less than or equal

floating point checks:
- **r4Is** check real4 is **NotNaN** / **Valid**
- **r8Is** check real8 is **NotNaN** / **Valid**

7.4.1 @if, @elseif, @else and @endif

Examples:

```
@if wParam gt 0
  @if lParam ne 3 || pMsg eq WM_COMMAND
    @if wParam ua 2 || lParam ub 5
       conout " Ain't Mabdelouahab's runtime comparisons great", lf
     @endif
  @elseif alor1 fGT uno && alor2 fNE dos
     @if alor3 r8Is NotNaN
        conout "  Is && working? Yes.  And is NotNaN", lf
     @else
        conout "  Is && working? Yes.  Ah... is NaN", lf
     @endif
  @elseif alor1 fGT 1.0 || alor2 fLT 4.0
      conout "  Is OR working? Yes", lf
  @else
      conout "  Nothing ", lf
  @endif
@else
    conout "  Nada ", lf
@endif
```

```
mov rax, 1
@if rax
      conout "  ----------------------------------------", lf
      conout "  TEST used ", lf
      conout "  ----------------------------------------", lf
@endif
```

```
mov r10, 1
mov r11, 2
@if r10 ub r11
      conout "  ----------------------------------------", lf
      conout "  CMP used ", lf
      conout "  ----------------------------------------", lf
 @endif
```

7.4.2 Others

Under development

7.5 Helper macros to translate 32 to 64 bits

These macros are a complement to make neutral bitness code. They are usefull in exactly same circunstances that requiere pseudo push/pop macros

7.5.1 Registers 32 to 64 bits Structured

A big problem in code translations from 32 bits to 64 bits it's related to X64 ABI Calling Convention. RCX and RDX have an specific function in 64 bits that ECX and EDX never have in 32 bits.

Beside, more simple replacement of registers, like XCX that become ECX in 32 bits and RCX in 64 bits, sometimes result incomplete because 32 bits registers can be used like DWORD or like POINTER, and they are same thing in 32 bits but have differents sizes in 64 bits.

But in 64 bits you have more registers. Then this macro make a systematic replace of registers to use exclusive 64 bits registers instead of RCX and RDX, and prefix "__" or "_" declare if register is used like DWORD or POINTER.

Example:

```
  @reg32_64  ecx,  r10
  @reg32_64  edx,  r11

ZAbs proc public USES _ecx lpSrc1:XWORD, lpDest:XWORD, uID:DWORD

      mov   __ecx, uID
      and   __ecx, Z_SINGLE or Z_DBL   ;check uID for operand size
      .if   ZERO?
            mov   __ecx,10
            jmp   @F
      .elseif __ecx == 4
            jmp   @F
      .elseif __ecx == 8
            jmp   @F
      .endif
      xor   eax,eax                 ;error code
      ret                           ;no valid operand size

       *   *  *
      ret

ZAbs endp
```

They are ecx/r10d and ecx/r10, or edx/r11d and edx/r11, for DWORD and POINTER in 32/64 bits.

7.5.2 Registers 32 to 64 bits Free Size

This macro is similar to previous one but simpler, older but still in use in some code translations, and prefix is "__". Usually used for DWORD but not limited to that.

Example:

```
  @reg32_64d cx, r11w
  @reg32_64d dl, r12b

  mov __cx, 15
  mov __dl, ah
```

7.5.3 Complementary macro of "m2m" macro

The "m2m" macros are used to move a variable to another variable. In 32 bits this can be achieved using a register or just push/pop, but in 64 bits only using a register. Because most "m2m" macros know how to proced accord to the presence or not of an auxiliar register, for neutral bitness auxiliar register must disapear if code was using push/pop in 32 bits. The idea is to use an exclusive 64 bit register.

Example:

```
m2m [xsi].orden, xordenmodelo, @if64bits(r12)
```

8 Accesory macros

Additionals macros are in Accs directory:

1 - Accesories macros useful for programming applications but not required by SmplMath.

- macros.inc: support macros already in Masm32 SDK macros.inc

- string.inc: support macros

- ConstDiv.inc: integer operations with inmmediates (32-bit)

2 - Macros required for ML64, like invoke macro. These are part of Masm64 SDK (see Masm32.com)

- Macros64G.inc: Invoke macro here can parse "OFFSET" argument. Optionally can process arguments promotions.