Quick Start

This document is a quick introduction for using the SmplMath macros with MASM and jWasm. For a full description see the main documentation. At the end of this document you will find two complete example programs with comments.

For the sake of simplicity, the following document assumes the usage of the MASM32 SDK[1] with MASM or jWasm[2] as Assembler.

Introduction

The SmplMath system is a collection of macros for evaluating mathematic expression as known from HLLs like c/c++, c#, … .There is a whole bunch of macros for expression evaluation, comparison and FPU related topics. The following examples show some of them:

Evaluation

```
fSlv y = a*x^2 + b*x + _c
mov eax, @fSlvI( 100*sin(2*pi*f*t+p) )
```

Comparison

```
.if fEQ(a,b) || fGT(x,1.E-7) || fLE(1,y)
   …
```

FPU settings

```
fpuSetPrecision ,REAL8
```

Installation

Copy the folder ***macros*** to the root directory of your working drive (mainly “C:\”).

If you won’t use this location, you must adjust the relative paths in \SmplMath*math.inc* (not recommended!!)

Requirements

The macros are tested with MASM version 6-11 and jWasm. However, using MASM version 8+ and the latest release of jWasm is recommended. The following example use the MASM32 SDK version 11.

Syntax

Here a quick list of the most important macros:

macro syntax	description	default precision
fSlv [dest] = *expression*	procedural macro	REAL8, SDWORD
fSlv4/8/10 [dest] = *expression*	procedural macro	REAL4/8/10, SDWORD
@fSlv4/8([dest=] *expression*)	returns a REAL4/8	REAL4/8, SDWORD
@fSlvI([dest=] *expression*)	returns a SDWORD	REAL4, SDWORD
fEQ/NE/LT/GT/LE/GE(a , b)	returns a Boolean byte (0,1)	REAL8, SDWORD
[ ] = optional		

The term ‘precision’ refers to, how to created constants and code. This means that the macros don’t affect FPU’s precision control.

[1] http://masm32.com/
[2] http://www.japheth.de/JWasm.html

Expressions can consist of sums (***+ -***), products (**_*/_**), exponents (***x^y***) and function calls. Also brackets can be used – there is generally no nesting limit, but extreme usage may cause the synthesis to fail (e.g. no more free registers). The priority of operators from high to low is:

signs, brackets, exponents, products, sums

<u>Please take care that signs has the highest priority</u>, which is problematic in combination with the exponentiation. e.g. $-x^2$ is treaded as $(-x)^2$. Use brackets to solve such problems.

Operands can be variables of any FPU supported data type (except BCD). If arithmetic for Addressing is need, it must place inside square brackets:

```
fSlv x = SDWORD ptr [esi+4*ecx] + y
fSlv x = myreal4[edi*8] + 2.3
```

Immediately values (constants) can be written either as integer or as floating point values (syntax=MASM specific).

The GPRs (eax,ax,al,ah,…) can also be used.

Functions can called by their name, followed by a bracket enclosed, comma separated list of arguments:

```
sin(x)   or    logbx(b,x)
```

For all current available function see the documentation or top of the file *\Macros\SmplMath\math_functions.inc*.

The fSlv-macros won't change any FPU setting, thus it is programmers job to setup the FPU. The precision setting for the SmplMath macros affects only how code and constants are created. However, there are some helper-macros like *fpuSetPrecision*, which change the FPU settings.

Generally each of the code producing macros has its own default precision setting for integer and floating point constants. This setting can either be changed by adjusting the global settings or by local usage of Attributes. These are passed in a comma separated list, enclosed by braces:

{attr1,attr2,[,...]}

Each Expression can have one randomly placed Attribute-list - it is recommended to place it at the end or begin of an expression. The following example shows how to change the local precision to SWORD and REAL4:

```
fSlv x = 1*2+123.0^0.5     {i2,r4}
```

Currently reserved Attributes for local precision control are:

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

Runtime behavior

The macros assume that the FPU stack is unused, so that all 8 FPU register can be used. Also they doesn't change the FPU settings – this must done explicit (e.g. by using *fpuSetPrecision*).

The GPRs are generally not violated, except if they are specified as destination operands. The Flags may be violated.

If no local variables are supplied through *fslvTLS()*, global variables are created in the _BSS Segment (.data?) if needed. This cause the macros to throw warnings that the produced code is not thread save. However, this only affects the function-like macros. The syntax for fSvlTLS() is:

```
foo PROC ...
    LOCAL fSvlTLS(name,cb)    ; name is optional. cb specific the number of bytes to allocate (default=16).
```

The fSlv-macros assume to be called from the ***.code***-section.

Further capacities

With version 2.0.0 of the SmplMath system, a front end / back end architecture has been introduced and support for x86-32 and x86-64 (jWasm only).

Currently there is a back end for the FPU, which is loaded by default for x32. Also there is a SSE2 back end, which is the default for x64 programs.

The second example shows how to switch to the SSE2 back end.

Example program 1 (MASM32 SDK, console application, FPU)

```
include \masm32\include\masm32rt.inc
.686p   ;\
.mmx    ; } needed!
.xmm    ;/

include \macros\smplmath\math.inc        ; include the macros
.code
main proc
LOCAL x:REAL8,y:REAL4,a:REAL4,b:REAL4,_c:REAL4
LOCAL sz[128]:CHAR
LOCAL fSlvTLS() ; <= share some locals with the fSlv-macros (thread save).
                ;    This call allocates 16 Bytes on the stack.

    ;/*-------------------------------------*/
    ;/* init FPU and set precision to REAL8 */
    ;/*-------------------------------------*/
    finit
    fpuSetPrecision ,REAL8


    ;/* load locals (constants only!) */
    ldl x=3.5, a=1, b=5, _c=1

    ;/*--------------------------------------*/
    ;/* let's do some basic calculations ;-) */
    ;/*--------------------------------------*/
    fSlv y = a*x^2+b*x +_c
    print "a*x^2+b*x +_c = "
    print real4$(y),13,10

    ;/* use SIB addressing */
    lea edx,y
    xor ecx,ecx
    fSlv x = REAL4 ptr [edx+4*ecx] * 123

    ;/*-------------------------------------*/
    ;/* @fSlvI() returns a SDWORD           */
    ;/* In this case it is a local variable */
    ;/*-------------------------------------*/
    ldl y = 12.34
    print str$( @fSlvI(y^2+2+pi) ),13,10    ; 'pi' is a known constant

    ;/*-------------------------------------------------------------*/
    ;/* In the following @fSlv8() returns the local x, because it's */
    ;/* type is the same as the return-type of the macro.           */
    ;/*-------------------------------------------------------------*/
    print real8$( @fSlv8(x = x^2) ),13,10

    mov sz[0],0
    print cat$(ADDR sz,"the logarithm of 12 to the base 3 is: ",real8$( @fSlv8(x= logbx(3,12) ) )),13,10
    mov sz[0],0
    print cat$(ADDR sz,"3^",real8$(x)," = ",real8$( @fSlv8(x= 3^x))),13,10

    ;/*---------------------------------------------------------------------------------------------------------*/
    ;/* The next line show the usage of comparison macros. Each of them returns a Boolean BYTE variable (0,1).  */
    ;/* This allows to call more than one macros per .IF-line. The returned byte is either a local, or if no    */
    ;/* local storage available, a global variable (_BSS).                                                      */
    ;/*---------------------------------------------------------------------------------------------------------*/
    .if fEQ( 1 , 2.E-7 ) || fGT( @fSlv4(x^2) , y )
            print "whatever",13,10
    .endif

    ;/*--------------------------------------------*/
    ;/* This example shows how to use Attributes   */
    ;/* to force new (local) precision settings    */
    ;/* {i2,r4} -> SWORD and REAL4                 */
    ;/*--------------------------------------------*/
    fSlv y = 1*2+3*4.0 +_c {i2,r4}

    .if fEQ( 1.2 , x {r10})  ; 1.2 will be a REAL8 constant
            ;...
    .endif

    inkey
    exit
main endp
end main
```

Example Program 2 (MASM32 SDK, console application, SSE2)

This example requires MASM version 7+ or jWasm.

```
include \masm32\include\masm32rt.inc
.686p   ;\
.mmx    ; } needed!
.xmm    ;/

include \macros\smplmath\math.inc        ; include the macros

;/**
; * fSlvSelectBackEnd searches and loads a back end that supports
; * specified instruction sets. Because the current SSE2 back end
; * also use the FPU (in some cases), the macro will throw a warning
; * that more instruction sets used than wished. This warning
; * can be suppressed by adding the token FPU.
; */

fSlvSelectBackEnd SSE2

; fSlvSelectBackEnd SSE2,FPU

.code
main proc
LOCAL x:REAL8,y:REAL8,a:REAL8,b:REAL8,_c:REAL8
LOCAL sz[128]:CHAR
LOCAL fSlvTLS()

    ;/**
    ; * With the macro fSlvVolatileXmmRegs you can control
    ; * which XMM registers are used by the evaluation macros.
    ; * The following call makes the XMM registers XMM4-7
    ; * non-volatile. This means they are not touched by the
    ; * fSlv-macros. Remarks that at least one register needed
    ; * for evaluation ;-D
    ; */
    fSlvVolatileXmmRegs remove,xmm4,xmm5,xmm6,xmm7 ; other operation on the list are: "add", "set" and "default"

    ldl x=3.5, a=1, b=5, _c=1


    ;/**
    ; * Remarks that using variables that are not typed according
    ; * to the current precision settings, force the macros to
    ; * make a conversion using the CVTxx2yy instruction.
    ; * For the fSlv macro, the default precision is REAL8.
    ; */
    fSlv y = a*x^2+b*x +_c
    print "a*x^2+b*x +_c = "
    print real8$(y),13,10

         ;/* use scalar compare instructions (COMSD) */
    .if fLT( 5 , 2.E7 )
        print "5 LT 2.0E7",13,10
    .endif


    inkey
    exit
main endp
end main
```