UASM v2.49

Extended User Manual

1. Introduction

JWasm has been an incredible assembler, especially for software development under Windows and one which we have used for research, personal and commercial projects. We believe strongly that it has a place in a modern development environment and can be used to create high-performance functions, libraries (which can be used from higher level languages) as well as be a compelling alternative for the development of entire solutions.

With the loss of support and development on JWasm as well as the desire to expand the existing product we have taken it upon ourselves to continue the product's evolution under a new name Uasm. We have tried to remain true to its roots and as such are continuing from the existing version numbering and reflect its origin in the name.

As the core usage, command lines parameters and descriptions of existing functionality remains unchanged from JWasm we include all of the original JWasm documentation and examples and describe only the new or enhanced features in this document.

2. New or Enhanced Features

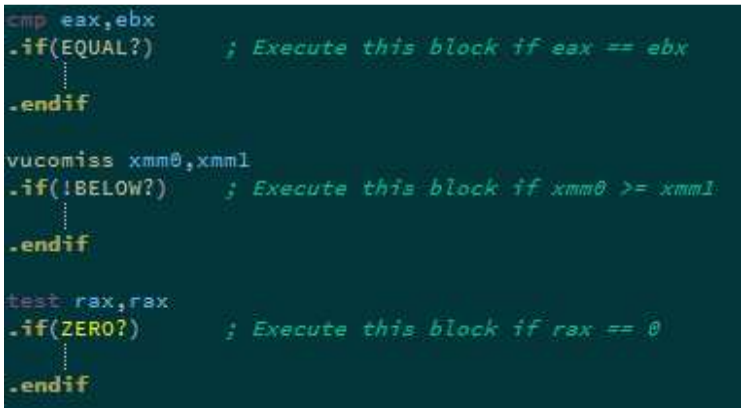
2.1) Flag-based comparison predicates

The flag-based comparison predicates allow you to generate higher level .IF syntax which will generate the corresponding Jcc instructions based on the condition type and any previous comparison. This allows the higher level syntax to be used with floating point and SIMD based comparison instructions.

Comparison Predicate	Signed/Unsigned	Equivalent Branch
LESS?	Signed	JGE <label>
GREATER?	Signed	JLE <label>
ABOVE?	Unsigned	JBE <label>
BELOW?	Unsigned	JAE <label>
CARRY?	N/A	JNC <label>
EQUAL?	Both	JNE <label>
ZERO?	N/A	JNZ <label>
OVERFLOW?	N/A	JNO <label>
SIGN?	N/A	JNS <label>

Comparison predicates can also be combined with a ! (NOT) in-front.

Example(s):



2.2) HLL .FOR / .ENDFOR loop

The syntax has been updated to use : instead of | as a separator as per previous implementation in JWasm. The code generation has been optimised and now supports a wide range of initializers, modifiers and conditional operators:

Initializers	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=,++,--,<constant>
Conditional operators	==,!=,> ,< ,>=,<=,&&,||,&,!,ZERO?,CARRY?,SIGN?,PARITY?,OVERFLOW?,LESS?,GREATER? ,ABOVE?

```
.for (edx=88, ecx=4 : eax != 24 && hWnd > lParam || ebx <= 20 || ebx >= 3 : eax=23, edx=24, ebx++)
    nop
.endfor

.for (rcx=0 : rcx<rdx : rcx++, rbx<<=4)

.endfor

.for (:r8:r8++,[rcx].RECT.top=eax)
    .if (rax)
        .continue
    .endif
    mov[rcx],dl
.endfor

; infinite loop
.for (::)
    .break .if eax
.endfor
```

One can use either .ENDFOR or the shorter .ENDF directive to end the for loop body.

2.3) Shadow Space Optimisations

Although the first four parameters are passed via registers, there is still space allocated on the stack for these four parameters. This is called the parameter homing space or shadow space and is used to store parameter values if either the function accesses the parameters by address instead of by value or if the

The minimum size of this homing space is 0x20 bytes or four 64-bit slots, even if the function takes less than 4 parameters.

- When the homing space is not used to store parameter values, Uasm uses it to save non-volatile registers (as specified by the USES clause).
- If a procedure doesn't have invoke it will not unnecessarily allocate the homing space.
- If a procedure doesn't use locals a FRAME will not be created so you will not need to use:

```
OPTION PROLOGUE:NONE
OPTION EPILOGUE:NONE
```

Procedure declaration…

```
OPTION PROLOGUE:PrologueDef
OPTION EPILOGUE:EpilogueDef
```

2.4) INVOKE Optimisation

If any of the first 4 parameter values are FALSE, NULL or 0 they will now be generated via XOR instead of MOV reg,0. In addition of other parameters after the fourth are compatible they can be written using the same XOR'ed register(s).

invoke testproc5, NULL,FALSE,NULL, 0,0, rdx

```
000000013FB618F0 33 C9             xor       ecx,ecx
000000013FB618F2 33 D2             xor       edx,edx
000000013FB618F4 45 33 C0          xor       r8d,r8d
000000013FB618F7 45 33 C9          xor       r9d,r9d
000000013FB618FA 48 C7 44 24 20 00 00 00 00 mov       qword ptr [rsp+20h],0
000000013FB61903 48 89 54 24 28    mov       qword ptr [rsp+28h],rdx
000000013FB61908 E8 76 00 00 00    call      testproc5 (013FB61983h)
```

Note: RDX being reused for zero in the sixth parameter.

2.5) General Procedure Prologue/Epilogue Optimisation

Uasm will now automatically make the best use of the available shadow space entries, parameters, locals to not only maintain alignment but to avoid executing any unnecessary prologue code sequences. For example if parameters are unused they won't be saved to shadow space. If no invokes are used within the procedure, automatic stack reservation will not be applied and so on. This allows all optimal procedure forms to be generated using the standard PROC FRAME USES arrangement and syntax.

2.6) RIP Register Support

Examples:

displacement EQU 200h

mov ah,[rip]
mov rax,[rip+3]
mov rax,[rip+400h]
mov cx,[rip+128]
mov [rip+127],cx
mov [rip+displacement],rbx
mov rbx,[rip+displacement]
mov rax,[rip]
mov [rip+1],sil
cmp byte ptr [rip], 90h
lea rbx,[rip]
lea rax,[rip+2]
call qword ptr [rip+400]
push [rip]
push [rip+80h]
pop [rip]

2.7) .SWITCH

Uasm now supports generation of optimised switch statements using either .if .else for short cases, jump tables or non-recursive binary search.

Example:

```
mov eax,280
.switch eax
        .case 273
                mov  edx,273
                .break
        .case 280
                mov  edx,280
                .break
         .case 275
                mov  edx,275
                .break
         .default
                mov edx,0
                .break
.endswitch
```

A new option is provided which allows the tradition C style of switch requiring (.break) and ASMSTYLE which does not.

option SWITCHSTYLE : ASMSTYLE
option SWITCHSTYLE: CSTYLE

Multiple cases can be combined as with:

```
 mov eax, 184h
  .switch eax
  .case 179h,180h,1c5h,17bh,17dh,182h,184h,185h
    mov edx,1d5h
  .case 1d3h
    mov edx, 1d3h
  .case 1f4h
```

```
    mov edx, 1f4h
  .case 200h
    mov  edx, 200h
  .default
    mov edx, 0
.endswitch
```

.default case can be omitted if not required.

Switch supports immediate numeric values as well as character immediates such as 'A', 'AB','ABCD','ABCD1234' etc.

In addition there is also:

OPTION SWITCHSIZE:SIZE.

The default is 0x4000 and is limited to a maximum of 0x8000.
The purpose is to allow you to customise the performance/size trade of switch generated jump tables.

An alias of .**ENDSW** has been added for .**ENDSWITCH**

2.8) VECTORCALL

Vectorcall calling convention support has been added for 64bit Windows targets.

To specify that a procedure should use this calling convention instead of the usual FASTCALL a new language type specifier has been added which must be included on both the PROTO and PROC as follows:

TestVectorcallProc PROTO **VECTORCALL** a:__m128

TestVectorcallProc PROC **VECTORCALL** FRAME a:__m128

The vector call convention is slightly more complex than fastcall so attention should be paid to how arguments are passed. If any of the first 4 arguments are integers, they are passed in RCX-R9 as per fastcall. If any of the first 6 arguments are floating point they're passed in XMM0-XMM5. The home space is increased to store 6 entries. Vectorcall supports passing of SIMD types by value in register rather than by reference and these are passed in registers XMM n or YMM n depending on the argument position. Vectorcall also introduces the concept of HFA (homogenous float array) and HVA (homogenous vector array). These types are populated into any empty registers after integer/floating point and vector types have been dealt with. If there aren't sufficient free registers then the entire HFA or HVA is passed by reference.
An HFA is any structure containing 1-4 floats or doubles (REAL4 or REAL8).
An HVA is any structure containing 1-4 valid SIMD type vectors (so you can think of it as a matrix).

HFA and HVA types are populated into registers component wise, so a 3 element float HFA passed as a single argument to a procedure would put the first float element into XMM0, the second into XMM1 and the final one into XMM2.

Due to the fact that a 4 element HFA is technically also a SIMD type, UASM has to make provision to determine the intended application and as such we now provide an include file of standard SIMD

data types. We rely on the naming convention specifically to ensure that a vector is handled appropriately and not treated as an HFA. The types have been named as per the C/C++ convetion: __m128 (the union) with specifically typed variants __m128f, __m128i etc.

UASM will allow you to declare a PROC that a type, for example the union __m128 but will determine how to use on a per-invoke basis. So for example it would be completely valid to have a PROC declared that expects a vector type __m128f and then pass it a variable declared as a 4 element float HFA. In this instance the HFA will be treated as a vector by that procedure, so you can consider it automatic type coercion where compatible.

As with FASTCALL UASM will automatically optimise the prologue to remove any unused argument. Ideally to obtain the performance benefit intended with VECTORCALL one should use the arguments directly from registers where possible. In the C/C++ version the homespace for vectorcall arguments is unused, however in UASM we will populate the stack with HFA/HVA/Vector elements if the argument is referenced and the substitute the memory address of the structure into the homespace provided.

Uasm implements the Vectorcall convention in a slightly different way to C/C++ in that all parameters passed by reference point to their original variables, we call this Referential Vectorcall

or R-VECTORCALL. This is completely compatible with C/C++ however special attention must be made for the following:

1) When calling an Assembly language RVECTORCALL function from C:
   Any modification of a parameter will only modify the local copy and not the original.
2) When calling a C VECTORCALL function from Assembly language, the function may modify variables which are assumed to be local copies thus modifying your original, so in these cases a copy should be made prior to invoking the function.

This choice was made to ensure optimal efficiency when using RVECTORCALL functions written in UASM from UASM as referential passing is far more efficient that making repeated local copies of entities on the stack.

For further information on Vectorcall please refer to:

https://msdn.microsoft.com/en-us/library/dn375768.aspx

https://blogs.msdn.microsoft.com/vcblog/2013/07/11/introducing-vector-calling-convention/

A new example source has been included to demonstrate the use of these structures, types and calls.

2.9) String Literal Support

Wide character literal data can now be declared with:

```
awideStr dw "wide caption ",0
```

String literals can be used directly in INVOKE using “” or L””. For a string literal to be accepted as such, the corresponding procedure parameter must be defined as **PTR**. Any other type will expect a

character constant or numerical value. String literal support is switched off by default but can be enabled with : **OPTION LITERALS:ON**

Declaring wide string data with dw will only happen with OPTION LITERALS:ON and using command line switches –Zm or –Zne will disable this.

2.10) Integer to Real type promotion

Certain data declarations required a real number literal to be supplied, this included REAL4, 8 and STRUCT members. Integer values can now be used in place:

```
; Automatic type promotion from integer to float
aReal REAL4 2

; This is example of initializing a union with floats (first sub-type)
; using normal syntax as well as hjwasm 2.17 update to promote integer literal to float
myVec1 __m128 { < 1.0, 2.0, 3.0, 4.0 > }
myVec2 __m128 { < 1, 2, 3, 4 > }
```

This can be quite convenient when using zeroes or identity rows in vectors and matrices.

2.11) New UNION syntax to specify sub-type initialization values

Previously a union could only be initialized using elements compatible with the first type. To overcome this limitation especially when using SIMD type structures as included in the supplied xmmtypes.inc a specific sub field can be specified.

Given the structures and union as follows:

```
__m128f struct
        f0 real4 ?
        f1 real4 ?
        f2 real4 ?
        f3 real4 ?
__m128f ends

__m128q struct
        q0 QWORD ?
        q1 QWORD ?
__m128q ends

__m128 union
        f32 __m128f <>
        q64 __m128q <>
__m128 ends
```

The following is now possible:

```
; Hjwasm 2.22 enhanced union type (now allows direct specification of sub-type to use in initialisation):
myVec4 __m128.f32 { < 1.0, 2.0, 3.0, 4.0 > }   ; you can try .f33 and hjwasm will emit an error when testing for valid sub-type.
myVec3 __m128.q64 { < 0x1234, 0x5678 > }
```

2.12) Integrated MACRO library

Uasm now provides a library of built-in macros which will automatically re-target to the current architecture setting (either SSE or AVX). This library can and will grow over time but for now includes:

Name	32bit	64bit	Description
CSTR	x	x	Literal text inline for use with invoke. Usage: Invoke SomeFunction, CSTR("This is literal text")
WSTR	x	x	Same as CSTR but for wide string data. Both of these are provided for backwards compatibility with MASM as UASM features direct support for string literal data in invoke.
FP4	x	x	Declare a real data element of the specified size inline. Usage: Movss xmm0,FP4(2.3)
FP8	x	x	As above for Double instead of float.
FP10	x	x	Full 80bit precision data item for use with FPU.
RV	x	x	Insert return value into EAX or RAX in a nested INVOKE call.
MEMALIGN	x	x	Round a number up to the next multiple of number. Usage: MEMALIGN <reg>,<number>
LOADSS	x	x	Optimised and architecture setting dependant load of an immediate float value to a simd register. Usage: LOADSS xmm0, 2.2 Uses a temporary register (EAX/RAX).
LOADSD		x	Same as above for double sized data.
LOADPS	x	x	Same as LOADSS but broadcasts to all elements.
MEMALLOC	x	x	Allocate memory using C runtime malloc. This primarily to provide a cross-platform way to allocate memory to be used by the OO library. Usage: MEMALLOC <size>
MEMFREE	x	x	Release memory via C runtime. Usage:

			MEMFREE <ptr>
UINVOKE		x	(refer to 2.18 for example) – Automatic inline form of invoke that supports determination of return type.
R4P	x	x	(refer to 2.19 for example) – To be used in place of real4 or real8 ptr with floating point HLL comparison. Alias name AsFloat.
R8P	x	x	Alias name AsDouble.
MOV64	x	x	Move a 64bit immediate value to memory. Usage: Mov64 <someVariable>, <immediate64>
MOV128	x	x	Move 128bit value to memory Mov128 dst, immLo64, immHi64
MOVXMMR128	x	x	Load 128bit value to an xmm register Movxmmr128 dstReg, immLo64, immHi64
SLXMMR		x	Shift left xmm register by bit count SLXMMR xmmreg, cnt
SHIFTLEFT128		x	Shift memory location 128bit left by bit count SHIFTLEFT128 memAddr, cnt
SRXMMR		x	Shift right xmm register by bit count SRXMMR xmreg, cnt
SHIFTRIGHT128		x	Shift memory location 128bit right by bit count SHIFTRIGHT128 memAddr, cnt
GETMASK128	x	x	GETMASK128 xmmreg, fieldmask for field GETMASK128 xmmreg, recordmask for record
NOTMASK128	x	x	NOTMASK128 xmmreg, fieldmask for field NOTMASK128 xmmreg, recordmask for record

```
;=======================================================
; MOV64 - Moving 64 bit immediate value to a memory location
; usage:  MOV64  num2,102030405060708h
;=======================================================
; MOV128 - Moving 128 bit immediate value to a memory location
; usage:  MOV128  num2,0001020304050607h,08090a0b0c0d0e0fh
;=======================================================
; MOVXMM128 - Loads 128 bit immediate value to any xmm register
; the value has to be reverted EG:
; MOVXMM128 xmm0, 0f0e0d0c0b0a0908h,0706050403020100h
; MOVXMM128 xmm1, "redroeht","desreveR"
; XMM0 = 0F0E0D0C0B0A09080706050403020100
; XMM1 = 726564726F6568746465737265766552
; but when you store register in memory we will get it right
; 0x00007FF7431A52D8  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f  ................
; 0x00007FF7431A52E8  52 65 76 65 72 73 65 64 74 68 65 6f 72 64 65 72  Reversedtheorder
;=======================================================
; SLXMMR - Shift left 128 bit xmm register value
; the value has to be reverted EG: "redroeht","desreveR"
; cnt is limited to low 7 bits, 127 or 7Fh, to avoid overflow
; MOVXMMR128  xmm0, "redroeht","desreveR"
; MOVXMMR128  xmm1, "redroeht","desreveR"
; XMM0 = 726564726F6568746465737265766552
; XMM1 = 726564726F6568746465737265766552
; SLXMMR  xmm0,16
```

```
; SLXMMR  xmm1,72;
; XMM0 = 0000726564726F656874646573726576
; XMM1 = 00000000000000000726564726F6568
; 0x00007FF651025258  76 65 72 73 65 64 74 68 65 6f 72 64 65 72 00 00  versedtheorder..
; 0x00007FF651025268  68 65 6f 72 64 65 72 00 00 00 00 00 00 00 00 00  heorder.........
;=========================================================
; SHIFTLEFT128 - Shift left 128 bit memory value
; the value is in normal order EG; 12345678
; .data
; num4 db "The actual order"
; num2 db "The actual order"
;  lea rax,num4             ;pointer to memory in rax
;  SHIFTLEFT128  rax,16
;  lea rax,num2             ;pointer to memory in rax
;  SHIFTLEFT128  rax,72
;before:
;0x00007FF7B4A45000  54 68 65 20 61 63 74 75 61 6c 20 6f 72 64 65 72  The actual order
;0x00007FF7B4A45010  54 68 65 20 61 63 74 75 61 6c 20 6f 72 64 65 72  The actual order
;after:
;0x00007FF7F6E95000  65 20 61 63 74 75 61 6c 20 6f 72 64 65 72 00 00  e actual order..
;0x00007FF7F6E95010  6c 20 6f 72 64 65 72 00 00 00 00 00 00 00 00 00  l order.........
;=========================================================
; SRXMMR - Shift right 128 bit xmm register value
; the value has to be reversed EG: 87654321
;  MOVXMMR128  xmm0, "redroeht","desreveR"
;  MOVXMMR128  xmm1, "redroeht","desreveR"
;  SRXMMR  xmm0, 16
;  SRXMMR  xmm1, 0x48
; before shift:
; XMM0 = 726564726F6568746465737265766552
; XMM1 = 726564726F6568746465737265766552
; after shift:
; XMM0 = 64726F6568746465737265766552000
; XMM1 = 65737265766552000000000000000000
; stored in memory:
; 0x00007FF764DA5268  00 00 52 65 76 65 72 73 65 64 74 68 65 6f 72 64  ..Reversedtheord
; 0x00007FF764DA5278  00 00 00 00 00 00 00 00 00 52 65 76 65 72 73 65  .........Reverse
;=========================================================
; SHIFTRIGHT128 - Shift right 128 bit memory value
; the value is in normal order EG; 12345678
;  lea rax,num4             ;pointer to memory in rax
;  SHIFTRIGHT128  rax,16
;  lea rax,num2             ;pointer to memory in rax
;  SHIFTRIGHT128  rax,72
;=========================================================
```

The integrated Macro Library can be disable with command line switch **–nomlib** if required.

2.13) OPTION ARCH:{SSE|AVX}

This setting determines which instruction set should be used for any automatically generated code. This includes prologue, epilogue, invoke as well as the built-in macro library.

For example the LOADSS built-in macro would be coded under SSE as :

```
mov eax,floatLiteral
movd xmmReg,eax
```

But under AVX it would use vmovd instead.

This setting is also available through command line switches –archSSE and –archAVX.

The default setting is to use the SSE instruction set.

The currently selected architected is also available through the built-in variable @Arch.

2.14) SIMPLIFIED CODE GENERATION

When using STACKBASE:RSP, frame:auto, win64:11 are now implied. Procedures with default settings will automatically be generated as FRAME procedures.

Using STACKBASE:RBP (or not specifying at all as this is the default) will imply frame:auto and default procedures will be generated as FRAME procedures. Win64 options 1-7 are valid.

In both cases the first local is aligned 16 and both options now implement optimisations to store only used parameters to home space.

STACKBASE:RBP will automatically optimise away the frame-pointer if no parameters or locals are referenced.

If the procedure is a leaf procedure and makes no use of locals or further invokes the add/sub rsp instructions will also be optimised out automatically.

2.15) NEW LANGUAGE TYPES

Procedures can now be decorated with attribute SYSTEMV to generate a 64bit SystemV ABI call.

For example:

```
OPTION ARCH:SSE
nixproc PROC SYSTEMV FRAME USES rbx xmm0 arg1:qword, arg2:DWORD, arg3:REAL4
   mov rbx,arg1
   mov ecx,arg2

IF @Arch EQ 0

          movss xmm10,arg3

ELSE

          vmovss xmm10,arg3

ENDIF

    ret
nixproc ENDP
OPTION ARCH:AVX
```

This produces the correct invoke, prologue and epilogue for use on 64bit Linux and OSX.

In addition a new OPTION REDZONE:{YES|NO} is provided to enable or disable the use of SystemV ABI Red-Zone optimisation.

The language type BORLAND has also been added to support Delphi / Free Pascal style register based fastcall.

2.16) OPTION PROC Extensions

OPTION PROC now allows for the specification of prologue and epilogue macro in a single statement, as a form of short-hand for OPTION PROLOGUE, OPTION EPILOGUE combined. It also allows you to specify the types NONE and DEFAULT.

OPTION PROC:NONE
is equivalent to the pair:
OPTION PROLOGUE:NONE
OPTION EPILOGUE:NONE

and

OPTION PROC:DEFAULT
is equivalent to the pair:
OPTION PROLOGUE:PROLOGUEDEF
OPTION EPILOGUE:EPILOGUEDEF

For custom combinations use:

OPTION PROC:MyPrologueMacro,MyEpilogueMacro

2.17) Additional Built-In Variables

@ProcLine, indicates the current source code line number relative to the start of the current procedure.

@ProcName returns the name of the current procedure.

@Platform <0|1|2|3|4> indicating the currently platform target as WIN32, WIN64, ELF32, ELF64, OSX MACHO64 to support improved cross platform assembly.

```
   IF @Platform EQ 0
       xor eax,eax
        echo 'im 32bit windows'
    ELSEIF @Platform EQ 1
       xor rax,rax
       echo 'im 64bit windows'
    ELSEIF @Platform EQ 2
       xor ebx,ebx
       echo 'im 32bit linux'
    ELSEIF @Platform EQ 3
       xor rbx,rbx
       echo 'im 64bit linux'
    ELSEIF @Platform EQ 4
       xor rbx,rbx
       echo 'im 64bit OSX'
    ENDIF
```

@LastReturnType, indicates the last function return type following an invoke call. Values are:

```
enum returntype {
       RT_SIGNED = 0x40,
       RT_FLOAT = 0x20,
       RT_BYTE = 0,
       RT_SBYTE = RT_BYTE | RT_SIGNED,
       RT_WORD = 1,
       RT_SWORD = RT_WORD | RT_SIGNED,
       RT_DWORD = 2,
       RT_SDWORD = RT_DWORD | RT_SIGNED,
       RT_QWORD = 3,
       RT_SQWORD = RT_QWORD | RT_SIGNED,
       RT_REAL4 = RT_DWORD | RT_FLOAT,
       RT_REAL8 = RT_QWORD | RT_FLOAT,
       RT_XMM = 6,
       RT_YMM = 7,
       RT_ZMM = 8,
       RT_PTR = 0xc3,
       RT_NONE = 0x100
};
```

2.18) Procedure Return types and UINVOKE

It is now possible to specify the return data type on procedures and prototypes as follows:

```
myfuncR4 PROTO (REAL4) :REAL4

myfuncD  PROTO (DWORD) :REAL4

myfuncQ  PROTO (SQWORD) :REAL4

myfuncX  PROTO (XMMWORD) :REAL4
```

The return type must immediate follow PROTO and unlike arguments takes no name and doesn't use a colon. The return type must be specified inside brackets to separate it from the argument list on both proto and proc to ensure there is no syntax ambiguity for un-typed parameters.

The matching procedure definitions are then:

```
myfuncR4 PROC (REAL4) FRAME a:REAL4

myfuncD PROC (DWORD) FRAME USES rbx a:REAL4

myfuncQ PROC (SQWORD) USES rbx a:REAL4

myfuncX PROC (XMMWORD) a:REAL4
```

The return type must immediate follow PROC and precede any other attributes such as FRAME, USES. Just like PROTO it requires no colon or name and is specified in brackets.

With the return-type specified, the following is now possible:

```
        vcmpss xmm0, uinvoke(myfuncR4, xmm2), xmm1, 0

        mov eax, uinvoke(myfuncD, xmm2)

        cmp rbx, uinvoke(myfuncQ, xmm3)

        vmovaps xmm0, uinvoke(myfuncX, xmm2)
```

The UINVOKE Macro Library function makes use of the @LastReturnType variable and can correctly return the relevant register to match the return type of the procedure and calling convention, IE: RAX, XMM0, etc.

2.19) HLL Floating Point Comparisons.

.IF , .ELSEIF , .WHILE have been extended to support floating point type comparisons. .FOR however supports only the classic integer arguments as before.

Examples:

```
.if(xmm0 < FP4(1.5))
.endif

.if(xmm0 > FP4(2))
.endif

.if(xmm0 < floatvar1)
.endif

.if(xmm0 == floatvar2)
.endif

.if(xmm0 < [rdx])
.endif

.if(xmm0 == [rdx+rbx])
.endif

.if(xmm0 < xmm1)
.endif

.if(xmm0 > xmm1)
.endif

.if(xmm0 <= xmm1)
.endif

.if(xmm0 == xmm3)
.endif

.if(real4 ptr xmm0 < xmm1)
.endif

.if(xmm0 < real4 ptr xmm1)
.endif

.if(R4P(xmm0) < xmm1)
.endif

LOADSD xmm0, 1.0
LOADSD xmm1, 2.0
LOADSD xmm3, 1.0

.if(xmm0 < doublevar1)
.endif
```

```
.if(xmm0 == doublevar2)
.endif

.if(real8 ptr xmm0 < FP8(1.5))
.endif

.if(xmm0 < real8 ptr FP8(1.5))
.endif

.if(real8 ptr xmm0 < xmm1)
.endif

.if(real8 ptr xmm0 > real8 ptr xmm1)
.endif

.if(real8 ptr xmm0 <= xmm1)
.endif

.if(xmm0 == real8 ptr xmm3)
.endif

.if(xmm0 == r8p(xmm3))
.endif
```

2.20) New DEFINE and UNDEF directives

These two new directives are built-in to UASM and allow something which wasn't previously easy before to check for the existence of a defined item :

```
DEFINE USELITERALS

IF  USELITERALS
  OPTION LITERALS:ON ; Allow string literals use in INVOKE
  printf PROTO :PTR, :PTR
ELSE
  printf PROTO :PTR, :vararg
ENDIF

UNDEF USELITERALS
```

2.21) Improved RECORD directive and operators.

COLOR RECORD blink:1, back:3, intense:1, fore:3

Inline code can use up to 64bit record types directly.

```
mov    cl, COLOR<1, 7, 0, 1>
mov    cobalt.rc, COLOR<1, 7, 0, 1>
```

Additionally, 64bit records can be declared in the data section and manually referenced.

```
COLOR RECORD blink:8, back:8, intense:8, fore:8,blink1:8, back1:8, intense1:8, fore1:8
safir COLOR <224, 225, 226, 14, 224, 225, 226, 14>     ; E0E1E20EE0E1E20E
```

128bit records are also supported:

.data

terra MYREC128 <0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,00,1,1,0,0,1,1,0,
0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,
0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,
0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0>

2.22) New FRAMEOFS Operator

A new operator, frameofs has been added which will return the relative displacement to the current stack frame for any local or procedure argument and can be used in 32bit and 64bit as follows:

```
main proc c aParam:dword
local   dwWritten:dword
local   hConsole:dword

    mov eax,FRAMEOFS(dwWritten)
    mov eax,[ebp+FRAMEOFS(hConsole)]

    ;for comparison
    mov eax,hConsole
    mov ebx,dwWritten
    mov eax,FRAMEOFS(aParam)
    mov eax,aParam
```

For esp/rsp based procedures the frameofs value can be used directly, however in the case of an ebp/rbp based procedure one has to take into account the setup of the stack frame. In a normal proc this would involve a push {ebp|rbp}. If you have a procedure with no automatic prologue, 4 or 8 will needed to be manually added to account for the lack of stack-frame or ideally one should be manually created as locals or parameters are being referenced.

2.23) ARGIDX, ARGSIZE, ARGTYPE Operators

These three new operators are designed to be used inside a procedure to obtain information about a specified parameter.

ExampleProc PROC FRAME par1:qword, par2:dword

mov eax,ARGIDX(par2)
; will return 2 -> ARIDX returns 0 if no matching parameter, else 1 based index.

mov eax,ARGSIZE(par1)
; size in bytes of the specified parameter. (8)

mov eax,ARGTYPE(par1)
; returns the type of the parameter in accordance with the following table:

```
RT_SIGNED = 0x40,
RT_FLOAT = 0x20,
RT_BYTE = 0,
RT_SBYTE = RT_BYTE | RT_SIGNED,
RT_WORD = 1,
RT_SWORD = RT_WORD | RT_SIGNED,
RT_DWORD = 2,
RT_SDWORD = RT_DWORD | RT_SIGNED,
RT_QWORD = 3,
RT_SQWORD = RT_QWORD | RT_SIGNED,
RT_REAL4 = RT_DWORD | RT_FLOAT,
RT_REAL8 = RT_QWORD | RT_FLOAT,
RT_XMM = 6,
RT_YMM = 7,
RT_ZMM = 8,
RT_PTR = 0xc3,
RT_NONE = 0x100
```

2.24) Native SIMD Data Types

The following data types are automatically supported by Vectorcall based invoke and can be used freely with the improved union initialisation to provide improved support for working vectors and SIMD type data especially when passing by value instead of reference:

```
__m128 union
    f32 __m128f <>
    i8  __m128b <>
    i16 __m128w <>
    i32 __m128i <>
    d64 __m128d <>
    q64 __m128q <>
__m128 ends
```

```
__m256 union
    f32 __m256f <>
    i8  __m256b <>
    i16 __m256w <>
    i32 __m256i <>
    d64 __m256d <>
    q64 __m256q <>
__m256 ends
```

Examples:

MyVector __m128 <1.0, 2.0, 3.0, 4.0>
MyVector2 __m128.i16 <10,20,30,40,50,60,70,80>
MyVectorCallProc PROTO VECTORCALL aVec:__m128, bVec:__m128f

2.25) EVEX promotion

Vex encoded instructions can now be promoted to their EVEX equivalents for AVX512 by using the {evex} prefix before the instruction. The main purpose of this is to generate an EVEX instruction which only uses AVX type semantics (which would automatically be VEX encoded) to support clearing the higher part of a ZMM register as part of the operation.

2.26) EVEX Broadcast operator

UASM now supports the MASM compatible BCST operator as an alternative to the AVX512 standard {1toN} syntax.

3. OPTION WIN64 Extensions

When using OPTION WIN64:***n***, the bits of ***n*** define optional configuration parameters for code generation in 64bit mode. The bit fields are as follows:

```
W64F_SAVEREGPARAMS = 0x01, // 1=save register parameters in shadow space on proc entry
W64F_AUTOSTACKSP   = 0x02, // 1=calculate required stack space for arguments of INVOKE
W64F_STACKALIGN16  = 0x04, // 1=stack variables are 16-byte aligned; added in v2.12
W64F_SMART         = 0x08, // 1=takes care of everything (Uasm)
```

Japheth introduced W64F_STACKALIGN16 in v2.12 because he was storing locals in reverse order: the first local was stored the last, so it wasn't possible to have the first local aligned 16.

This has been changed it so that the first local is first after the stack homing area and is guaranteed to always be aligned to a 16 byte boundary.

With this feature it is possible to keep the LOCALS you need to have aligned to 16 bytes as the first ones and avoid using this bit flag.

So in general one would use a value of 11 for ***n*** *(or 15 if additional LOCAL alignment is required).*

- AVX (ymmword) or AVX512 (zmmword) LOCALS will be automatically aligned on the stack too.

4. OPTION EVEX

A new OPTION EVEX:{0|1} has been added to enable the assembly of AVX512 code and extended registers (ZMM0 – ZMM31, XMM/YMM 16-31).

5. OPTION ZEROLOCALS

A new OPTION ZEROLOCALS:{0|1} has been added to clear LOCAL values declared inside a PROC to zero. A threshold is set so that the generated code will use immediate moves if only a few locals are present, for larger local allocation on the stack string reps are used.

6. OPTION FLAT

For some types of low-level programming, especially OS kernel and boot-loader code the model adopted by some other non-MASM style assemblers lends itself very well to mixing code of different bit types, for example switching between real and protected or long mode.

To make this type of programming more straight-forward UASM implements a flat mode. A single code section is create with the normal .code simplified directive, however the new directives USE16/USE32/USE64 can be used on their own to switch the current code-generation mode.

The USExx directives can only be used inside the .code section.

In addition Flat mode will write segment data out in the order in which it is declared in the source, unlike simply writing out a normal source file to BIN output which follows the model's segment ordering.

Example:

```
option flat:1
.code

    org 0h
    USE16

    xor ax,ax
    xor eax,eax

    org 100h
    var1 dd 10

    USE32

    xor ax,ax
    xor eax,eax

    org 1000h
    var2 dd 20

    USE64
    xor ax,ax
    xor eax,eax
    xor rax,rax
```

7.UASM Object Oriented Language Extension

Introduction

Uasm introduces a set of language extensions made available through the built-in Macro Library System. One of these extensions is the ability to implement Object Orientation in Assembler Code.

The approach is slightly different from traditional OO in that it doesn't make use of inheritance but provides the concept of an interface which can be mapped to homogenous or even heterogeneous classes as long as they conform to the interface's layout.

As with C++ it is good practice to keep each class definition in its own file and the implementation in another. This keeps code clean, modular and allows different modules to share the definition of the class and any related types.

The OO library can be used as-is on Windows platforms and can be used on Linux / OSX with one caveat:

To provide memory allocation routines (and the implementation of MEMALLOC, MEMFREE macros) it is assumed that you have linked to libc and provided prototypes for malloc and free. As neither Linux nor OSX provide heap allocation routines as found on Windows, libc's malloc/free are used instead.

- ***See LIN64_2 example in the Samples folder***.

Declaring a Class

```
IFNDEF _CLASS_PERSON_
_CLASS_PERSON_ EQU 1

CLASS Person
	CMETHOD GetName
	CMETHOD SetName
	CSTATIC IsHuman
ENDMETHODS
	fname	db 128 DUP (?)
	age	db 0
	human	db 0 ;static variable
ENDCLASS

pPerson TYPEDEF PTR Person

ENDIF
```

As with C++ you should implement an inclusion guard in your class definition file through the use of IFNDEF.

A class is simply declared as CLASS <name> and ENDCLASS.

The class data shares a lot in common with a simple structure data type and thus allows member fields to be specified directly in the class definition.

Methods are purely named at this point using either CMETHOD (Instance method), CSTATIC (Static method.

Methods should be declared first before any fields and their end marked with ENDMETHODS.

It is often useful to also define a pointer to object type such as pPerson in this case.

The class directive creates a static copy of the object structure. This is used to store static elements as well as provide a means to directly invoke methods without going through a vtable. Each class technically creates two static backing structures, one for the vtable/methods and one for the static data and fields.

Each CMETHOD or CSTATIC entry creates not only the correct types, prototypes on the object but creates a relevant vtable entry for when the class is actually instantiated.

Implementing the Class

Implementation of Init (Constructor), Destroy (Destructor) methods is **mandatory**.

```
; Constructor -> Can take optional arguments.
;-----------------------------------------------------
METHOD Person, Init, <VOIDARG>, <USES rbx>, age:BYTE

	LOCAL isAlive:DWORD
	; Internally the METHOD forms a traditional procedure, so anything that you can
	; do in a PROC you can do in a method.

	; On entry into any method RCX is a pointer to the instance
	; and the correct reference type is assumed.
	mov [rcx].human, 1                      ; Hence this is possible.
	mov (Person PTR [rcx]).human, 1         ; Alternative forms of reference.
	mov [rcx].Person.human, 1               ; " "

	mov isAlive,0
	mov al,age
	mov [rcx].age,al

	.if( age < 100 )
		mov isAlive,1
	.endif
```

```
        ; Constructor MUST return thisPtr in rax (the implicit self reference
        ; passed in RCX).
        mov rax,thisPtr

        ret
ENDMETHOD

; Destructor -> Takes no arguments.
;-------------------------------------------------------
METHOD Person, Destroy, <VOIDARG>, <>
        mov [rcx].age,0
        ret
ENDMETHOD


; Return pointer to name.
;-------------------------------------------------------
METHOD Person, GetName, <qword>, <>
        lea rax,[rcx].fname
        ret
ENDMETHOD

; Set person name.
;-------------------------------------------------------
METHOD Person, SetName, <VOIDARG>, <USES rbx>, pNameStr:QWORD

        lea rsi,pNameStr
        lea rdi,[rcx].fname

copyname:
        mov al,[rsi]
        mov [rdi],al
        .if( al == 0 )
                jmp done
        .endif
        inc rsi
        inc rdi
        jmp short copyname
done:

        ret
ENDMETHOD

; Static method to check if a person is a human.
;-------------------------------------------------------
STATICMETHOD Person, IsHuman, <qword>, <>, somebody:PTR Person
        _STATICREF rax, Person
        mov al,(Person PTR [rax]).human
        ret
ENDMETHOD
```

Methods can specify an optional return type and USES clause, which can be left empty with <>. For a method which does not return any value and for the mandatory init and destroy (constructor and destructor methods) the return type should be specified as **VOIDARG**.

Be aware that under Windows 64bit calling convention, thisPtr is passed in RCX, while under SYSTEMV thisPtr will be supplied in RDI. This can be made platform-agnostic by using the built-in.

On Windows one can also declared methods using **VECMETHOD** and **STATICVECMETHOD** which make use of the VectorCall ABI. This isn't as relevant under SystemV as the default ABI already supports vector and SIMD parameters.

Declaring and Instantiating Objects

Objects are instantiated via the use of either the **_NEW** or **_RBXNEW** directives. Both can be in-lined into other expressions, statements and invokes.

Objects can ONLY be instantiated via the NEW operators, any declaration of a variable of an object type should be a pointer to that object, so objects cannot be statically declared as they wouldn't be correctly initialised and no constructor call would be made.

```
local myPerson:PTR Person
local age:BYTE
mov age,36

mov r10,_RBXNEW(Person, 10)
_DELETE(r10)

mov myPerson,_NEW(Person, age)
_DELETE(myPerson)
```

Instances can be deleted via **_DELETE**. For most basic uses simple LOCAL or GLOBAL variables can be used to store pointers to object instances.

To declare an array of objects you can additionally use :

```
mov rbx,_ARRAY(Person,8)
_DELETEARRAY rbx
```

This will attempt to create an array of references (pointers) to the objects, or for primitive types and normal structures a fully sized array in memory.

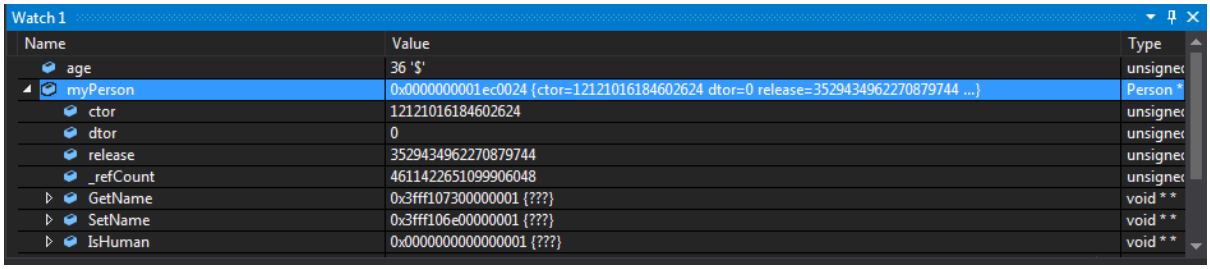
An alternative directive to LOCAL is supplied that supports < > in names. Technically this makes no difference in the code that is generated, however due to the assembler's parsing of quoted and literal text in macros it allows us to "simulate" higher level generic types in names, for example:

```
_DECLARE objArray[8], PTR Person
_DECLARE myList, PTR List<Float>

CLASS List<Float>
; Any class which has the same methods/parameters/ordinal is said to conform to an
interface. Methods can be invoked via the interface to enable RT-Polymorphism.
	CMETHOD Items
	CMETHOD AddItem
	CMETHOD RemoveItem
	CMETHOD Clear
	CMETHOD Trim
	CMETHOD Sort
```

```
        CMETHOD InsertItem
ENDMETHODS
        Count    dq 0
        Capacity dq 0
        CurIdx   dq 0
        itemsPtr dq 0
        itemSize dq 0
        itemType db 0
ENDCLASS

METHOD List<Float>, <QWORD>, <>, Init, count:QWORD, itemSize:VARARG
        mov byte ptr [rcx].itemType,LIST_FLOAT
        mov rax,4
        mov [rcx].itemSize,rax
        .if(rdx > 0)
                mov [rcx].Capacity,rdx
                imul rax,count
                invoke HeapAlloc,_oo_heap,0,rax
                .if(rax == 0)
                        THROW OUT_OF_MEMORY
                .endif
                mov rcx,thisPtr
                mov [rcx].itemsPtr,rax
        .endif
        mov rax,thisPtr
        ret
ENDMETHOD
```

Debugging

Debugging support is implicit due to all methods and members being fully typed, arguments are visible inside methods and entire object instances can be examined:



Invoking Methods

A number of accelerator macros are provide to call methods either directly, indirectly via their vtable entry or inline in other invokes including specified return types:

```
; Direct invoke (via the generated structure type):
_INVOKE List<String>,AddItem,myList,myString

; Indirect invoke via the object instance vtable:
_VINVOKE myString,String,Trim, FALSE

; Direct, with in-line direct _I call
_INVOKE String,ToLower,_I(List<String>,Items,myList,0),FALSE
```

_V, _VF, _VD, _VW, _VB are also provided to provide in-line vtable invocations that return a result of the specified type : V == QWORD, VF == float/real, VD == DWORD, VW == WORD, VB == BYTE.

NB: _I, _STATIC and **_V** inline methods fully support the defined return type of the method and can be used in place of the explicitly typed versions.

Methods may also be invoke using a c-style pointer syntax as follows:

```
; Method invocation using c-style calls.
person1->Calc(1.0)
.if( person1->Calc(1.0) == FP4(2.0) )
       xor eax,eax
.endif

; Using a register as an object pointer with c-style calls requires the type to
specified after the pointer.
; The register must be doubley-indirect, in that it points to the pointer to the
object, which allows for rapid iteration through lists of object pointers.
lea rsi,person1
[rsi].Person->Calc(1.0)

.if( [rsi].Person->Calc(1.0) == FP4(2.0) )
       xor eax,eax
.endif

xor rax,rax
lea rsi,person2
[rsi+rax].Person->Calc(1.0)
xor rax,rax
.if( [rsi+rax].Person->Calc(1.0) == FP4(2.0) )
       xor eax,eax
.endif

; Static method invocation using _STATIC and HLL style dot operator:

mov eax,_STATIC(Person, IsHuman, person1)              ; Old Inline Way...
```

```
_SINVOKE Person, IsHuman, person1                    ; Old Way...
Person.IsHuman(person1)                              ; New way...
```

Refer to the oo ASM sample for a more detailed example.

Interfaces

An interface is effectively a generic contract, which can be applied to invoke methods and access members of unrelated object instances as long as they conform.

An example of this in action is combined with < > support to implement a range generic container class types for List<int>, List<float>, List<double>.
By creating an IList interface we can access any of these types in a consistent manner.

```
OINTERFACE IList            ; Common Container Protocol/Interface.
       CVIRTUAL Items,<qword>, idx:QWORD
       CVIRTUAL AddItem, <voidarg>, objPtr:QWORD
       CVIRTUAL RemoveItem, <voidarg>, idx:QWORD, release:BOOL
       CVIRTUAL Clear, <voidarg>, release:BOOL
       CVIRTUAL Trim, <voidarg>
       CVIRTUAL Sort, <voidarg>
       CVIRTUAL InsertItem, <voidarg>
ENDMETHODS
ENDOINTERFACE
```

An interface definition begins with O**INTERFACE** <name> and ends with **ENDOINTERFACE**. Common methods are declared with the **CVIRTUAL** specifier. The first parameter for the **CVIRTUAL** definition is the return-type, which can be declared as **VOID** with <VOIDARG>.

Note however that virtual methods do specify their arguments as these generate only prototypes and no actual code. This is to ensure type-conformance.

The first entry on this particular interface and any classes which want to share this type specify an **Items** method.

This is a special method, which allows for accelerator macros to be used to access any object which implements some form of iterator (IE: a container).

For example, in the specialisation class List<float> we have:

```
METHOD List<Float>,Items, <voidarg>, <USES rbx>, idx:QWORD
       mov rax,[rcx].itemsPtr
       mov rbx,[rcx].Count
       .if(rdx < rbx)
              movss xmm0,[rax+rdx*4]
       .else
              THROW INDEXOUTOFBOUNDS
       .endif
       ret
ENDMETHOD
```

Which will then allow other code to access its internal collection with:

```
; This protocol interface is purely to allow for acceleration of typeless calls to get
; items out of classes that need an iterator / [] access.
OINTERFACE Iterator         ; Common Container Protocol/Interface.
       CVIRTUAL Items, <voidarg>, idx:QWORD
ENDOINTERFACE

; The direct specific way:
mov rax,_V(myList,List<Float>,Items,0)
; The generic iterator interface way:
mov rax,_ITEM(myList,0)
```

Interfacing with COM Objects

UASM 2.46 combines the new pointer type, HL calling syntax and OO framework components to allow simple use of COM object interfaces in much the same way as would be familiar to C/C++ developers.

For example the IDIRECT3D9 COM Interface can be declared as :

```
COMINTERFACE IDirect3D9
    CVIRTUAL RegisterSoftwareDevice, DWORD, :PTR
    CVIRTUAL GetAdapterCount, DWORD
    CVIRTUAL GetAdapterIdentifier, DWORD
    CVIRTUAL GetAdapterModeCount, DWORD
    CVIRTUAL EnumAdapterModes, DWORD
    CVIRTUAL GetAdapterDisplayMode, DWORD
    CVIRTUAL CheckDeviceType, DWORD
    CVIRTUAL CheckDeviceFormat, DWORD
    CVIRTUAL CheckDeviceMultiSampleType, DWORD
    CVIRTUAL CheckDepthStencilMatch, DWORD
    CVIRTUAL CheckDeviceFormatConversion, DWORD
    CVIRTUAL GetDeviceCaps, DWORD, Adapter:DWORD, DeviceType:DWORD, pCaps:PTR
    CVIRTUAL GetAdapterMonitor, DWORD
    CVIRTUAL CreateDevice, DWORD
ENDCOMINTERFACE

LPDIRECT3D9 TYPEDEF PTR IDirect3D9
```

Invocation of COM methods is then almost identical to C/C++ :

```
    mov direct3d,Direct3DCreate9( D3D_SDK_VERSION )
    .if( rax != 0 )
```

```
        direct3d->GetDeviceCaps( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, ADDR caps )
    .endif
```

Internally UASM identifies the type of object being called, for COM objects the double-indirection form is used to extract the vtable pointer from the object pointer when making the call, where as UASM classes keep the vtable in the instance itself.

8. Simple Symbolic Output Format

This document describes the file format generated by the –Fs command line switch.
The file is primarily intended for information purposes only or to provide a degree of self-hosting symbolic debug support when combined with the flat binary output option. This is particularly useful for embedded, pre-kernel and kernel module code.

Offset	Size (bytes)	Description
0	4	**Symbol Count**
4 .. n	1	**Symbol type**: 'S' = segment 'T' = type 'P' = proc 'L' = address/location
.	4	**Offset (from segment start)**
.	4	**Size**
.	4	**Type:** MT_SIZE_MASK = 0x1F, /* if MT_SPECIAL==0 then bits 0-4 = size - 1 */ MT_FLOAT = 0x20 MT_SIGNED = 0x40, /* bit 6=1 */ MT_BYTE = 1 - 1, MT_SBYTE = MT_BYTE | MT_SIGNED, MT_WORD = 2 - 1, MT_SWORD = MT_WORD | MT_SIGNED, MT_DWORD = 4 - 1, MT_SDWORD= MT_DWORD | MT_SIGNED, MT_REAL4 = MT_DWORD | MT_FLOAT, MT_FWORD = 6 - 1, MT_QWORD = 8 - 1, MT_SQWORD= MT_QWORD | MT_SIGNED, MT_REAL8 = MT_QWORD | MT_FLOAT, MT_TBYTE = 10 - 1, MT_REAL10= MT_TBYTE | MT_FLOAT, MT_OWORD = 16 - 1, MT_YMMWORD = 32 - 1, MT_ZMMWORD = 64 - 1, MT_PROC = 0x80, /* symbol is a TYPEDEF PROTO, state=SYM_TYPE, typekind=TYPE_TYPEDEF, prototype is stored in target_type */ MT_NEAR = 0x81, MT_FAR = 0x82, MT_EMPTY = 0xC0, MT_BITS = 0xC1, /* record field */ MT_PTR = 0xC3, MT_TYPE = 0xC4, /* symbol has user-defined type (struct, union, record) */ MT_SPECIAL = 0x80, /* bit 7 */ MT_SPECIAL_MASK = 0xC0, /* bit 6+7 */ MT_ADDRESS = 0x80, /* bit 7=1, bit 6 = 0 */
.	N bytes	Name, null terminated.

9. C-Style Calling

UASM supports a higher level form of procedure invocation based on other common higher level languages such as C. Obviously UASM is NOT a full compiler with register allocator so there are some caveats when using this style of calling. Firstly function calls may only be nested one level deep as with MyFunction(OtherFunction(), 1, 2). These nested functions will preserve the original values of the outer call, especially in the case of SystemV or x64 Fastcall to ensure registers are preserved. To avoid potential register conflicts in high level expressions such as .IF, only a single c-style call is allowed. UASM will emit an error if either of these conditions are broken to avoid potential misuse of the feature. As with other procedures in UASM you are freely able to specify the return type, and all invocations of functions, OO methods respect the return-type information allowing you to directly use calls in expressions that are dependent on the return type being something other than <R|E|AX>.

The C style calling convention also supports the use of the (Address-Of) operator & which is functionally identical to using ADDR.

Here are some examples of ways in which the calls can be made, directly, nested, in a high level expression or as an operator as part of a memory address or other expression.

```
.if ( MyProc6( MyProc8() ) == 7 )
   xor eax,eax
.endif

mov eax,MyProc(10, 20)
lea rsi,myVar
mov eax,[rsi + MyProc12()]
mov myVar,MyProc(11,12)
mov myVar2,MyProc12()
vmovaps xmm1,MyProc5(1.0)

MyProc3()
MyProc(10,ADDR MyProc)
MyProc2("this is a literal")

.if( MyProc(10,20) == 30 )
   xor eax,eax
.endif

.if( MyProc5(1.0) == FP4(2.0) )
   xor eax,eax
.endif

.if ( MyProc6(7) == 7 )
   xor eax,eax
.endif

MyProc(MyProc3(), 20)
```

```
MyProc4(&MyVariable)

MyProc(10, MyProc4())
MyProc(MyProc3(), MyProc2("stringy"))

MyProc7( 10, 20, myVar, myVar2, 2.0, 30 )

MyProc( MyProc( MyProc3(), 10 ), 20)

.if ( MyProc6( MyProc8() ) == 7 )
   xor eax,eax
.endif

MyProc7( 10, 20, myVar, myVar2,     MyProc9( 10, 20, myVar, myVar2, 2.0, 30 ), 30 )
```

10. OPTION HLCALL:{ON|OFF}

The default for this option is ON allowing all forms of C style calling and pointer based method invocation. It can be switched to OFF if there are specific cases where the expansion may cause undesirable results, such as a macro parameter expecting a raw input in the form **procname().**

11. Supported Calling Conventions / Language Types

SYSCALL
C
STDCALL
PASCAL
FORTRAN
BASIC
FASTCALL
VECTORCALL
SYSVCALL
DELPHICALL

12. Building with Visual Studio

Copy the Uasm targets archive contents to:

C:\Program Files (x86)\MSBuild\Microsoft.Cpp\v4.0\V120\BuildCustomizations

(**Note**: the version of the folder and exact location will depend on your version of Visual Studio). **You should put Uasm.exe in this folder:**

C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin\x86_amd64

(Note: Rename Uasm32.exe or Uasm64.exe to Uasm.exe when copying into this location).

13. Support and Contact

For any information, queries and general assembly language guidance please visit the MASM32 forum and Uasm related boards at : http://www.masm32.com/board/index.php

14. Thank You

We'd like to thank everyone in the assembler community for continuing to support the language and related products and hope you enjoy using Uasm!

A few people deserve special mention:

Agner Fog for providing ObjConv and assisting with bug fixes during the testing of EVEX encodings.
Japheth for providing JWasm and all his years of effort on the Assembler.

Happy Coding!

Branislav Habus and John Hankinson