Rudy's Delphi Corner

A Tale Of Two Assemblers

This is a preliminary version, a work in progress. I will probably (have to) add, enhance or rewrite some parts.

Download

Introduction

Recently, in the Embarcadero public forums, there was a vivid discussion about the announcement that the first incarnation of a Win64 Delphi compiler would very likely not have a built-in assembler (BASM). The advise was to use an external assembler instead, and it was said that Embarcadero would probably choose NASM, the open source Netwide Assembler.

***UPDATE**: It is clear now, that there is a Win64 built-in assembler in Delphi XE2 and above. You will not have to use NASM, at least not at the moment.*

There were complaints that the lack of a built-in assembler would mean that the compiler would be rather useless, and that converting code to an external assembler like NASM would be too much work. To check this, I decided to rewrite my Decimals.pas unit, which uses BASM throughout, entirely or at least partly in NASM, to see how feasible this is and how much work it would be.

In this process, I learned a lot about the similarities and differences between BASM and NASM, and managed to write a small package of macros that could make the conversion a little easier. This article decribes what I experienced during this process.

NASM

NASM is a versatile, but rather simple assembler. It does accept the usual Intel assembler syntax like `MOV EAX,EDX` or `LEA EAX,[EBP+SomeOffset]`, but it does not accept large parts of the syntax of Microsoft's MASM (ml.exe or ml64.exe), nor does it accept most of the syntax of Borland's TASM's Ideal Mode. Of course it also doesn't know Delphi's syntax for comments and other Delphi features you can use in Delphi BASM, like *VMTOffset*, etc.

For what it's worth: for this article I used version 2.09.03 from 27 Oct. 2010.

NASM can produce a range of output formats. It can produce the most usual 16 bit, 32 bit and 64 bit object files, as well as simple .bin or .com files. It took me a while to find out how to create Delphi-compatible 32 bit OMF object files (see my article about using object files with Delphi). There are two things that must be done: the chosen format must be *obj*, and each and every segment declared in the source file must be declared as *USE32*. The first segment declaration should be close to the top, otherwise you'll get an empty 16 bit segment, which makes your object file unusable.

NASM is a command line compiler. It comes with a console setup. I enhanced it to have a window of 160 characters wide, 75 characters high and Lucide Console 14pt as font. I wrote a little batch file (*asm.bat*) to compile the decimals.asm file:

```
@echo off
```

```
nasm -fobj -Ox -l%1.out %1.asm
```

It is used as:

```
asm decimals
```

The options I use are:

- `-fobj` — Sets output format to OMF. This can contain 16 and 32 segments!
- `-Ox` — Full optimization. Where possible, chooses the smallest opcode/literal combination.
- `-l%1.out` — %1 is the first argument of the batch file, e.g. *decimal*. `-l` gives the name of the listing file
- `%1.asm` — The file to be assembled, e.g. *decimal.pas*

If there are no errors, this generates the files *decimals.obj*, in OMF format, and the text file *decimals.out*, which is the listing file, showing what code has been generated. I usually have both files (*decimals.asm* and *decimals.out*) open in the Delphi editor. The *decimals.out* file will be automatically updated by the IDE.

Writing assembler for NASM

As I already said, every segment must explicitly be declared as `USE32`, otherwise it will be generated as 16 bit. The NASM documentation says you can also use `[BITS 32]` (probably near the top of the file), but I did not have any success with that.

The decimals.asm file I have has three segments (or sections) declared. I don't know if all three are necessary, but it works:

```
1  section data    public use32
2
3  ; data declarations, e.g. records or external data
4  ; (external to this file, e.g. in a Delphi unit)
5
6  section const   public use32
7
8  ; constant declarations
9
10 section code    public use32
11
12 ; code
```

> *As you can see, both are declared as* `use32`*. Note that NASM is not case sensitive, except for its declared labels and "variables", which can be case sensitive or case insensitive, depending on how you declared them. You can also see that comments start with a semicolon, instead of* `//`*.*

Data

I declared all data in the *data* section of the .asm file.

Records

In *decimals.pas*, I declare a few types. To be able to use them, similar structs must be declared in NASM. In NASM, there is a standard macro, called `struct`, which allows you to do that. But I had a few problems with the built-in `__SECT__` macro used there, so I wrote my own `record` macro which doesn't use `__SECT__`. It can be found, together with a few other macros and definitions I prepared, in the *delphi.mac* file that accompanies this article.

One example is the *Decimal* type itself. If you omit the many methods and operator overloads, this is what remains, in Delphi:

```
type
  Decimal = packed record
  private
    Lo: Longword;               // Hi:Mid:Lo form 96 bit unsigned mantissa
    Mid: Longword;
    Hi: Longword;
    case Byte of
      0: (Reserved: Word;       // always 0
          Scale: Shortint;      // 0..28
          Sign: Byte);          // $80 = negative, $00 = positive
        1: (Flags: Longword);
  end;
```

Here follows the translation to NASM. Note that NASM does not "memorize" any declared sizes (it only uses the size to reserve space, but does not automatically generate opcodes for a certain operand size), so I cheated a bit and declared *Flags* as *Word*, which allowed me to declare *Scale* and *Sign* too:

```
record  Decimal
.Lo             resd    1
.Mid            resd    1
.Hi             resd    1
.Flags          resw    1
.Scale          resb    1
.Sign           resb    1
end;
```

The "fields" of such a record can then be accessed like:

```
        MOV     EAX,[ESI+Decimal.Hi]
        MOV     [.MyVar+Decimal.Hi],EAX
```

The *record* macro actually declares an absolute segment (like the CGA screen segment `0B800H` in the old DOS days) at 0, and the "fields" are in fact local labels (that is why they start with a dot). Fortunately, this is similar to how you can address fields of a Decimal in BASM too (BASM knows a few more ways, but if you want to stay compatible with both syntaxes, you can use this syntax, which both assemblers can understand).

The *end* at the end of the record declaration is a macro too. I found it nicer than a specific *endrecord*. It can end a *record* declaration as well as a *function* or *procedure* declaration, and will generate code appropriate for where it is placed. The semicolon is unnecessary (it just starts a comment), but makes it look a little nicer, IMO.

Similarly, I declared the *TAccumulator* type. In Delphi, this is a variant record. I have not found a way to declare such records in NASM yet, so I declared an alternative *TAccumulator2* which maps to the original record the same way, but uses the alternative layout.

Alignment

You should be aware that such record declarations are always packed. There is no alignment whatsoever. If you need aligned records, you can take care of the alignment by yourself, by using the appropriate `resb` (byte), `resw` (word), `resd` (dword), etc. directives. They are explained in the NASM documentation.

If you have a record like:

```
{$A8}
type
  TTest = record
    B: Boolean;
    L: Longint;
  end;
```

then it is not necessary to reserve *B* (or actually *.B*) as a byte, since the assembler will not use the declared size for its opcode generation anyway. You can just as well reserve it with `resd`, since that will take care that the *Longint* is properly aligned on a 4 byte offset:

```
record  TTest
.B      resd    1
.L      resd    1
end
```

This means that you must know the details about record alignment and know how to pad with bytes to achieve a certain alignment. This is a lot easier in Delphi and BASM, of course.

Another possibility is to use the built-in `align` and `alignb` macros. One example of the use of `alignb` can be seen in the *TFormatSettings* record:

```
record  TFormatSettings
.CurrencyString                 resd    1
                                alignb  1
.CurrencyFormat                 resb    1
                                alignb  1
.CurrencyDecimals               resb    1
                                alignb  2
.DateSeparator                  resw    1
                                alignb  2
.TimeSeparator                  resw    1
                                alignb  2
.ListSeparator                  resw    1
                                alignb  4
.ShortDateFormat                resd    1
                                alignb  4
.LongDateFormat                 resd    1
                                alignb  4
.TimeAMString                   resd    1
                                alignb  4
.TimePMString                   resd    1
                                alignb  4
.ShortTimeFormat                resd    1
                                alignb  4
.LongTimeFormat                 resd    1
                                alignb  4
.ShortMonthNames                resd    12
                                alignb  4
.LongMonthNames                 resd    12
                                alignb  4
.ShortDayNames                  resd    7
                                alignb  4
.LongDayNames                   resd    7
                                alignb  2
.ThousandSeparator              resw    1
                                alignb  2
.DecimalSeparator               resw    1
                                alignb  2
.TwoDigitYearCenturyWindow      resw    1
                                alignb  1
.NegCurrFormat                  resd    1
end;
```

> *The documentation for NASM says that* `align` *should be used for code and data segments, while* `alignb` *should be used for bss segments (bss segments contain uninitialized data — using* `resb` *etc., while data segments contain initialized data — using* `db` *etc.). The record declarations only contain* `resb`, `resw`, *etc. declarations, so the proper alignment is done using* `alignb`, *since that uses* `resb` *by default to reserve space, while* `align` *uses sequences of* `NOP` *or other assembler code to fill the gaps, which is not allowed in a data (or bss) segment.*

That still means you'll have to know how alignment works (Delphi uses so called natural alignment — automatically padding variables so they begin on addresses that are a multiple of their size), but it allows you to align records properly

External data

In this context, "external" means that the data are not defined in the .asm file. They must be defined in the Delphi

program. In the *decimal.pas* unit, I define a few typed constants that are accessed by the assembler routines. This kind of data must be declared `extern` in the .asm file:

```
extern          PowersOfTen
extern          MaxMultiplicands
extern          MaxMultiplicandsMid
```

In the .pas file, they are declared thus:

```
const
  // ...
  HLW = High(Longword);
  HUI64 = High(UInt64);

  PowersOfTen: array[0..9] of Longword =
  (
             1,
            10,

    // values snipped

    1000000000
  );

  MaxMultiplicands: array[0..9] of Longword =
  (
    HLW,
    HLW div 10,

    // values snipped

    HLW div 1000000000
  );

  MaxMultiplicandsMid: array[0..9] of Longword =
  (
    Longword(HUI64),
    Longword(HUI64 div 10),

    // values snipped

    Longword(HUI64 div 1000000000)
  );
```

Declaring them as extern made them readily accessible from the NASM file.

True constants and enums

In *decimals.pas*, I declare a few constants and one enumeration, in a `const` section. True constants do not take up memory, they are mere symbols, so they must be re−declared in the assembler file. A few examples:

```
const
  SingleMShift   = 8;
  SingleMMask    = $007FFFFF;
  SingleEShift   = 23;
  SingleEMask    = $FF;
  SingleBias     = $7F;

  DoubleMShift   = 11;
  DoubleMMask    = $000FFFFF;
  DoubleEShift   = 52;
  DoubleEMask    = $7FF;
  DoubleBias     = $3FF;

  ExtendedEShift = 64;
  ExtendedEMask  = $7FFF;
  ExtendedBias   = $3FFF;
```

The translation is:

```
section const   public use32

SingleMShift    equ     8
SingleMMask     equ     $007FFFFF
SingleEShift    equ     23
SingleEMask     equ     $0FF
SingleBias      equ     $7F

DoubleMShift    equ     11
DoubleMMask     equ     $000FFFFF
DoubleEShift    equ     52
DoubleEMask     equ     $07FF
DoubleBias      equ     $03FF

ExtendedEShift  equ     64
ExtendedEMask   equ     $7FFF
ExtendedBias    equ     $3FFF
```

In NASM, you can also use the `$` sign to declare a hex literal, but as `$` can also be the first character of a macro-local label, it must be followed by at least one numeric character. If you forget the `0` in `$0FFF`, you will probably get an error about an unknown label `$FFF`.

Enumeration values are true constants too. I wrote a simple macro that allows you to define simple enums, i.e. the ones that start at ordinal 0 and do not give any numerical values to their members (I am not familiar enough with NASM macros to be able to make it also accept the more complicated ones). Here is the Delphi type:

```
type
  TDecimalErrorType = (detOverflow, detUnderflow, detZeroDivide, detInvalidOp,
    detParse, detConversion, detInvalidArg, detNaN);
```

and here is the NASM translation using the `enum` macro:

```
enum detOverflow, detUnderflow, detZeroDivide, detInvalidOp, \
  detParse, detConversion, detInvalidArg, detNaN
```

(the `\` character is the line continuation character in NASM)

The enum macro does not declare the *TDecimalErrorType*, as that is not needed in assembler anyway (it can be declared as *DWORD*, if necessary), but it declares the numerical constants it defines.

Differences in versions

One big problem appeared when I had finally converted the second overload of *TryParse*, the overload that is passed a *TFormatSettings* parameter. Errors started happening.

The first problem was that implementing *TryParse* as *Decimal.TryParse* in NASM caused an internal error in the linker. This was caused by the fact that there are two overloaded versions of *TryParse*. So I had to rename the routine as *Decimal.TryParseWithSettings* and call that from the second overload of *Decimal.TryParse*. The internal error was gone, but it still produced bad results.

The strings I used to create *Decimals* in my test routines were generated by other code, and since this was on a German Windows, the “decimal point” was represented by a comma. e.g. I had code like:

```
A := '31415926,535897932384626433833';
B := '10,0000000000000000001';
C := '8,5467';
```

The comma is the proper decimal point character on this version of Windows, but I had taken the *TFormatSettings* declaration from *SysUtils.pas* of Delphi 2010. The code was, however, compiled in Delphi XE. It took me a while to find out that *TFormatSettings* was completely redefined in Delphi XE! So I had to take the different *TFormatSettings*

declaration and convert that. You can see the result in *Decimals.asm*.

I have not found a really good solution for such problems. It would be good if there were complete NASM conversions of the data structures, enums, etc. used in Delphi, supplied with Delphi (or, say, by JEDI. That would mean you'd only have to set the proper directories and automatically load the correct declarations for your version of Delphi.

Code

The most important part of *decimals.pas* is of course its code. Most of it is in assembler. I used the built-in assembler, but to test NASM, I made a copy, called it *Decimals_nasm.pas* and moved the BASM routines to *decimals.asm*, using the Delphi editor.

I first tried a simple routine, without local variables and without any stack variables. The routines are in Delphi's `register` calling convention. The first parameter is passed in *EAX*, the second in *EDX* and the third in *ECX*. This means the routine does not need any special stack handling, i.e. no prolog code to set *EBP* to the stack frame, or code to change *ESP* to allocate local variables. It also doesn't need epilog code to clear the stack:

```
class function Decimal.Div192by32(Quotient: PLongword; const Dividend: Decimal;
  Divisor: Longword): Longword;
asm
        PUSH    ESI
        PUSH    EDI

        MOV     EDI,EAX                 // EDI is Quotient
        MOV     ESI,EDX                 // ESI is Dividend
        CMP     ECX,0
        JNE     @NoZeroDivide32

        MOV     EAX,detZeroDivide
        JMP     Error

@NoZeroDivide32:

        // Some code snipped

        MOV     EAX,EDX

        POP     EDI
        POP     ESI
end;
```

This is the orginal translation:

```
; class function Decimal.Div192by32(Quotient: PLongword; const Dividend: Decimal;
;  Divisor: Longword): Longword;

global Decimal.Div192by32

Decimal.Div192by32:

        PUSH    ESI
        PUSH    EDI

        MOV     EDI,EAX                 ;// EDI is Quotient
        MOV     ESI,EDX                 ;// ESI is Dividend
        CMP     ECX,0
        JNE     .NoZeroDivide32

        MOV     EAX,detZeroDivide
        JMP     Error

.NoZeroDivide32:

        ; same code snipped

        MOV     EAX,EDX

        POP     EDI
        POP     ESI
```

```
27          RET
```

I defined a few macros, `procedure`, `begin` and `end`, later also `asm` and `function`, which made the code look like this:

```
1   ; class function Decimal.Div192by32(Quotient: PLongword; const Dividend: Decimal;
2   ;  Divisor: Longword): Longword;
3
4   function Decimal.Div192by32
5
6   asm
7
8           PUSH    ESI
9           PUSH    EDI
10
11          MOV     EDI,EAX                 ;// EDI is Quotient
12          MOV     ESI,EDX                 ;// ESI is Dividend
13          CMP     ECX,0
14          JNE     .NoZeroDivide32
15
16          MOV     EAX,detZeroDivide
17          JMP     Error
18
19  .NoZeroDivide32:
20
21          ; same code snipped
22
23          MOV     EAX,EDX
24
25          POP     EDI
26          POP     ESI
27  end;
```

As you can see, the Delphi local label `@NoZeroDivide32` is translated as the NASM local label `.NoZeroDivide32`. Comments starting with `//` are converted to start with a semicolon. And the macros have a similar meaning as the corresponding keywords in Delphi. `Error` is a procedure in the Delphi code, and was declared as `extern` in the code section of the NASM source.

The `end` macro only generates a `RET` opcode. In the next section, you can see that it can do more than that.

> *Note that the name of the function,* `Decimal.Div192by32`*, can be declared exactly as it would be declared in a Delphi file. This is a fortunate circumstance.*

Stack parameters and local variables

The code in *decimals.pas* also contains functions that have parameters that are passed on the stack, for instance parameters of type *Single*, *Double* or *Extended*, and local variables. I wrote a set of macros, `param` and `var`, that work together with the `procedure` or `function`, `begin` or `asm` and `end` macros. An example of their use follows. This is the original (the code is rather long, so I snipped most of it):

```
1   class procedure Decimal.InternalFromExtended(out Result: Decimal;
2     const Source: Extended);
3   var
4     A: TAccumulator;
5     LResult: ^Decimal;
6   asm
7           PUSH    ESI
8           PUSH    EDI
9           PUSH    EBX
10          MOV     LResult,EAX
11
12          XOR     EAX,EAX
13          MOV     A.L0,EAX
14          MOV     A.L1,EAX
15          MOV     A.L2,EAX
16          MOV     A.L3,EAX
17          MOV     A.L4,EAX
18          MOV     A.L5,EAX
```

```
        MOV     A.Flags,EAX
        MOV     EDI,DWORD PTR [Source]
        MOV     ESI,DWORD PTR [Source+4]
        MOVZX   EDX,WORD PTR [Source+8]

        // code snipped

        MOV     EAX,A.Flags
        MOV     [EDI].Decimal.Flags,EAX

        POP     EBX
        POP     EDI
        POP     ESI
end;
```

This is how the macros are used:

```
; class procedure Decimal.InternalFromExtended(out Result: Decimal;
;   const Source: Extended);

procedure Decimal.InternalFromExtended

param   Source,Extended

var     A,TAccumulator
var     LResult,Pointer

asm
        PUSH    ESI
        PUSH    EDI
        PUSH    EBX
        MOV     [.LResult],EAX

        XOR     EAX,EAX
        MOV     [.A+TAccumulator.L0],EAX
        MOV     [.A+TAccumulator.L1],EAX
        MOV     [.A+TAccumulator.L2],EAX
        MOV     [.A+TAccumulator.L3],EAX
        MOV     [.A+TAccumulator.L4],EAX
        MOV     [.A+TAccumulator.L5],EAX
        MOV     [.A+TAccumulator.Flags],EAX
        MOV     EDI,DWORD PTR [.Source]
        MOV     ESI,DWORD PTR [.Source+4]
        MOVZX   EDX,WORD PTR [.Source+8]

        ; code snipped

        MOV     EAX,[.A+TAccumulator.Flags]
        MOV     [EDI+Decimal.Flags],EAX

        POP     EBX
        POP     EDI
        POP     ESI
end;
```

`procedure Decimal.InternalFromExtended`

This sets up a new NASM context called `_procedure_` and initializes all procedure/function related local variables

`param Source,Extended`

This declares a stack variable. It generates a local define (local "variable") called `.Source` and reserves 12 bytes on the stack for it. The second parameter of the `param` macro must be a known type with a known size. This can either be a type declared in the *delphi.mac* file, or one generated by a `record` definition.

`var A,TAccumulator`

`var LResult,Pointer`

This works similar to `param`, but it makes sure that *ESP* is changed to make room for local variables. These can be accessed via a negative offset to *EBP*. Both macros, `var` and `param` will cause the generation of prolog and epilog code, which is similar to the code generated by Delphi.

`asm`

If local variables or stack parameters were declared, it generates a prolog setting up *EBP*. If local variables were declared, it also generates code that changes *ESP* to make room for them:

```
        PUSH    EBP
        MOV     EBP,ESP
```

```
3           SUB     ESP,%$DELPHI_varsize_
```

end

If local variables or stack parameters were declared, it will generate code to reset *EBP* to its original state. If stack parameters were declared, it also generates the necessary `RET n` opcode, otherwise it only generates a simple `RET`. Of course, it only does that in the `_procedure_` context. In a `_record_` context, it generates different code. Finally, it returns to the previous context:

```
1           MOV     ESP,EBP
2           POP     EBP
3           RET     %$DELPHI_paramsize_-8
```

As you can see, the macros make defining a function or procedure in a little easier than when you'd have to use pure NASM syntax.

Note that the macros had to be written thus, that local labels are generated. So a variable called `A` *will become* `.A` *and a parameter called* `Source` *becomes* `.Source`*. I first had them generate global defines, but that caused huge problems (like infinite recursion) when an identifier was redeclared using* `var` *or* `param`*, since the preprocessor would expand the macro parameter* `A` *into its previous definition (e.g.* `EBP-36`*)* before *it was passed to the macro, as well as* inside *the text of the macro. The macro would make things worse or cause an error, or both. A manual* `%undefine` *of the identifier would prevent that, but that can easily be forgotten. I wanted the macros to take care of most of the work Delphi BASM does too.*

If someone with better knowledge of NASM's macro language has a good solution for this, one that allows the use of global names (i.e. names without a leading `'.'`*) while avoiding the recursion problems, I'd love to hear about it.*

NASM vs. BASM

It is clear that NASM is not BASM. The built-in assembler is certainly more convenient to use than an external assembler like NASM. But someone who wants to use the external assembler (for whatever reason, like a lack of BASM in a future version) must know about some important differences. Here are some of the most important differences someone used to Delphi's BASM must know about. Some of them were touched in the previous text already.

Instruction syntax

There are quite a few syntax differences. The NASM assembler is rather simple (the creators call that a feature, but I am not so sure about that). So called effective addresses only know one syntax: they must be fully enclosed in square brackets. So this:

```
1  asm
2          ...
3          MOV     EAX,[ESI].Decimal.Hi    ;// ESI points to a Decimal
4          MOV     D.Hi,EAX                ;// D is local var of type Decimal
5          MOV     D.Sign,0
6          ...
7  end;
```

Can only be coded as this:

```
1          ...
2          MOV     EAX,[ESI+Decimal.Hi]            ; [ESI+8]
3          MOV     [.D+Decimal.Hi],EAX             ; [EBP-16+8]  = [EBP-8]
4          MOV     BYTE [.D+Decimal.Sign],0        ; [EBP-16+15] = [EBP-1]
5          ...
```

In the example above, you can see a few other differences:

- There is no difference between variables and labels, and certain types of labels/variables (local, macro local, context local, etc.) *must* start with certain character combinations. I would have preferred another way to distinguish such types (e.g. by keyword or placement) as by a name prefix (or suffix). It is also the reason why a `var` or `param` macro produces a local label/variable, i.e. one that starts with a dot, like `.D` in the example above.
- Syntax like `[ESI].Decimal.Lo` or `MyVar[EBX]` is not allowed. As I said above, the alternative syntaxes to define an effective address Delphi knows are not allowed in NASM. This is a restriction and less flexible, but it has the advantage that it is far less ambiguous (for the reader) what is actually assembled.
- Simple dot syntax for members of a type, like `.D.Lo` or `[.D.Lo]` is not possible: the "type" (actually, the segment name, since that is what the `record` macro uses) must be mentioned: `[.D+Decimal.Lo]`.
- The size or type of a variable or member is not regarded by the assembler. Even if *Decimal.Sign* is declared with `resb`, and `D` is declared as `Decimal`, you must still specify the byte access with `BYTE` or `@BYTE PTR`. But if the compiler can deduce from the other operand what you are storing, you can omit the size specification, e.g. `MOV [.D+Decimal.Sign],AL` does not require a `BYTE` specification.

The syntax `BYTE PTR`, `WORD PTR`, etc. is not allowed in NASM. You must use `BYTE`, `WORD`, etc. That is why I put the empty define `PTR` in *delphi.mac*. Now you can use `BYTE PTR`, etc. just like in your BASM code. Note that `PTR` is not required in BASM either, so if you want to make (new) code compatible, don't use it.

Comments are different. In NASM, there are only single line comments that start with a semicolon (`;`) and end at the end of the line. The Delphi syntax `//` is regarded a numerical operator in NASM. To make new code more easily convertible to NASM, you can use the combination `;//` to start a comment in both syntaxes. In Delphi, this is seen as a statement separator `;` followed by a comment `//`, while in NASM it is simply seen as a comment `;`.

Case sensitivity for labels/variables is the default. You can use the preprocessor commands `%idefine` and `%ixdefine` to define case insensitive labels/variables, but the NASM pseudo-instruction `EQU` seems to be case sensitive. `EQU` has the advantage that you really define a constant (which can not be inadvertently redefined) and not just some preprocessor text.

> *Note that instructions, like* `MOV EAX,EDX` *are* not *case sensitive. So if you are used to writing those in lower case, or mixed case, there is no need to convert that to upper case. Only variables/labels are generally case sensitive.*

Instructions are single line by default too. If you want them to spill into the next source line, you must use the `\` continuation character. This applies to the preprocessor too.

Miscellaneous

If things are hard to do in assembler — this applies to BASM too —, like raising an exception or setting the length of a dynamic array, or accessing properties of another object, it makes sense to do it in a simple routine in Delphi and call that from assembler. In *decimals.pas*, you can see that I do that in the routine Error().

The same applies for things that are much easier in Delphi's BASM, e.g. accessing members of an object, or calling virtual methods. Inheritance and hidden or private members make it almost impossible to re-define objects in full in NASM, so the best you can do is to write such methods in Delphi as stubs that pass all required data to the assembler routine and pass back a return value, if there is one.

The same applies to virtual or dynamic methods. NASM has no way to access `VMTOFFSET` or similar, so if the external assembler must really call a virtual or dynamic function in an object, write a private routine that calls the virtual function and the assembler code calls that function instead, passing all data as required. This is more tedious and adds one extra calling level, but it is probably not something that is required often.

Conversion

I mentioned most of the things that must be done to convert BASM to NASM already. Simple syntactical text conversions like changing `[ESI].Decimal.Sign` to `[ESI+Decimal.Sign]`, changing `@MyLabel` to `.MyLabel` or comments

from `//` to `;//` can easily be done using the GREP capabilities of the Delphi IDE. The rest must be done manually, but using the macros *delphi.mac*, this should not be too time consuming.

Conversions of function frames, possibly with local variables and stack parameters, can be done using the *delphi.mac* file I wrote. That file is far from perfect, as I am not too familiar with the NASM preprocessor yet, but the macros made it a lot easier to convert my BASM functions to NASM.

Delphi.mac also provides a way to declare records or enums. I am sure these macros can be improved as well, but they do — more or less — what I wanted.

The Delphi side

The Delphi side is quite easily explained. The object file must be linked in using `{$L}` or `{$LINK}`:

```
{$L decimals.obj}
// Alternatively, you can use:
{$LINK 'decimals.obj'}
```

Preferrably, the *.obj* file is in the same directory as the *.pas* file, but it can also be in a different directory. To access it, you can use relative addressing like `'..\asm\decimals.obj'` or absolute addressing like `'C:\source code\asm\decimal type\decimals.obj'`.

Routines done in external assembler must be declared `external`:

```
class procedure Decimal.InternalFromExtended(out Result: Decimal;
  const Source: Extended); external;
```

Unlike you would do with routines in a DLL, you do *not* declare module, `name` or `index` (e.g. `external 'bla.obj' name 'Decimal.InternalFromExtended'`). The linker is responsible for finding the routines. If it can't, it will report an error. It (and probably NASM too) will also report an error if the *.obj* file requires a routine to be defined in the Delphi program.

There is, as far as I know, no way to declare overloads in assembler, so overloads should be done in Delphi, and simply contain calls to assembler routines with different names:

I already mentioned it before: routines have the same name in NASM as they would have in Delphi, so there is no need to use global ("undotted") names and call these from a stub in Delphi, you can directly use fully qualified member names.

The future

Due to a lack of a Delphi for 64 bit, I was not able to test the 64 bit assembler or ELF64 generation (the format that is likely to be chosen, according to *Allen Bauer* of Embarcadero). I don't know if there may be specific things to be aware of when doing this. I know that exception and stack handling are different, but I assume that the compiler will know how to take care of them. I usually leave exception handling to Delphi anyway.

If BASM is really omitted from one or more future versions of Delphi, it would be crucial to have at least better support for the use of NASM or any other suitable external assembler. Some of the following points would make this a lot easier:

- A compiler directive that would provide automatic support of an external assembler like NASM, just like this is done for resource files.
- NASM conversions of the records, enums, etc. in the Delphi runtime library in the form of include files, more or less like the *.hpp* files generated for C++Builder.
- Passing of certain information like directories, conditional defines, Delphi version, etc. to the NASM assembler in the proper NASM command line format.
- A new page in the project options for the use of NASM (or any other assembler), which allows the user to set up directories for the assembler, a search path for include files and assembler files, conditional defines,

post- and prefixes for file names, etc.
- A way to call “primitives”, IOW private runtime routines in *System.pas* like `System.UStrFromPWCharLen@` that you can call from BASM but not from plain Delphi code. No BASM would mean no access to those primitives at all, not from Delphi and not from external assembler.

Conclusions

It is possible to use Delphi without a built-in assembler. But it is not nearly as convenient, and poses quite a few problems.

Code written in BASM syntax must be converted to NASM syntax. This is far more elementary, but IMO most of it can be done using a few *grep* search and replace actions.

Every data type used in Delphi must be converted to NASM syntax. This can be done manually by the users — it *must* probably be done manually by the users for their own types like records and enumerations. It can also be done by, say, Embarcadero or a group like JEDI. Fact is that it hasd to be done and poses quite a lot of work, especially if these types differ between versions, like the *TFormatSettings* example described above.

Overloaded assembler functions are not possible. Functions can be overloaded in Delphi, but they can’t be in an external assembler. The only way to deal with this when using an external assembler is to make the overloaded functions call differently named functions in the external assembly. IOW, the overloads are mere wrappers for the real assembler functions, which must have different names.

One fortunate feature of the Delphi/external assembler system is that you can name methods implemented in NASM the same way as you would in Delphi. A function like *Decimal.InternalMultiply* can also be called the same way in NASM (unless it is overloaded, of course).

Existing code must be converted to NASM syntax. Most of this can be done using the *grep* capabilities of the Delphi editor. I hope that the few macros and type declarations I put together in *delphi.mac* make some other parts a little easier as well. But it is still work that has to be done.

But for a 64 bit Delphi, large parts of the code must be rewritten anyway, and that is probably more work. If this rewrite is immediately done in the proper NASM syntax (a syntax which BASM supports as well, except for some small things like local labels starting with ‘.’ instead of ‘@’), the extra time for using an external assembler is possibly negligible compared to the time required to rewrite the code.

Access to classes is a real problem. It is very easy in BASM, but quite a problem in any external assembler. There is no proper way to mimic inheritance, so access to private, protected or public fields of a class instance is almost impossible to do in a more or less transparent way. The only way to circumvent it is to write extra static class methods or global routines that contain the assembler parts, to which the instance methods pass the proper information as parameters in Delphi code.

Another problem with classes is that an external assembler has no access to pseudo-macros like *VMTOFFSET* or *DMTINDEX*, so calling virtual or dynamic methods is also impossibly done transparently. Virtual methods must probably be called by extra written wrappers that call the real code (IOW, to call a virtual method from assembler, you write a non-virtual method in Delphi that does nothing but call the virtual method with the proper parameters and pass any results back, and that non-virtual method is called by the assembler code).

Version problems are no problem in BASM, but another big one in external assembler. Proper support by Delphi, as described in the proposals for the future above, is the only useful way to deal with this.

In BASM, you can access certain private functions in *System.pas* using a special syntax, e.g. the routine *_UStrFromPWCharLen* in *System.pas* can be accessed as `System.@UStrFromPWCharLen` from assembler. This is not possible in external assembler, and it is, without BASM, impossible to call them from Delphi code too (at the moment — it would be nice if the compiler could give access to such “primitives” from Delphi code). In the case of `System.@UStrFromPWCharLen` I wrote a high level equivalent which probably implicitly calls the actual primitive. This requires some thought and knowledge about the internals of these routines and can be pretty difficult to do in high level code.

Routines using type info require wrappers in Delphi too, but the information is extremely hard to access from external assembler.

There are probably other problems with access to high level features not used in *Decimals.pas* and *Decimals_nasm.pas*, like generics, anonymous methods, RTTI, etc. If I find some, I will discuss them here.

Finally

While it is possible to use an external assembler, it takes quite some extra effort to do a *conversion*. For a *rewrite* or *completely new code* the difference between BASM and external assembler is far less dramatic. OK, it requires quite a few wrappers, but with access to primitives, and using the proper syntax, it should be doable. Support from the compiler and the IDE would, IMO, be a necessity, though. Especially access to NASM conversions of the runtime types is crucial, but also IDE support in the form I described is required.

Last update: Feb. 20, 2019