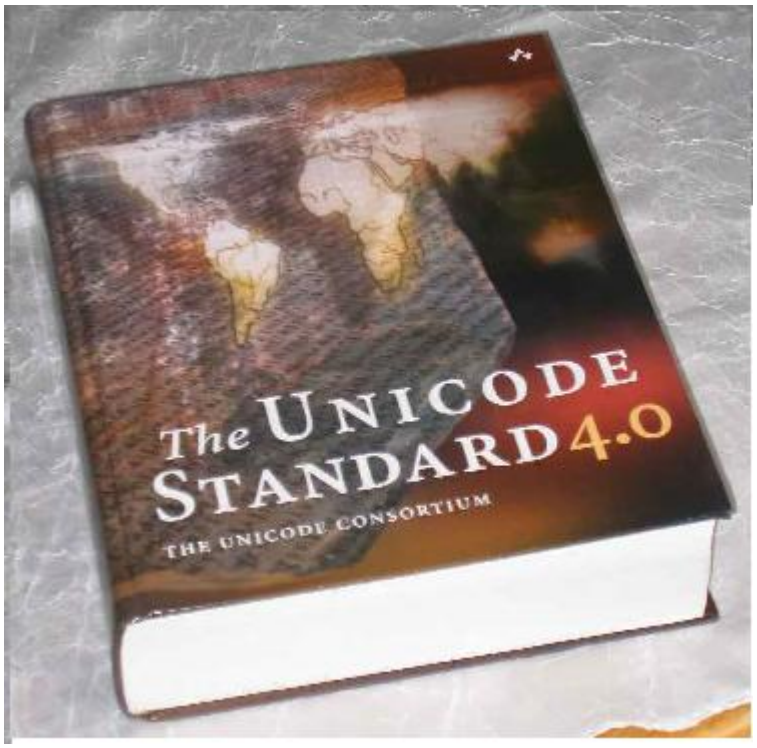
The new string unicode type

Marco van de Voort

December 3, 2010

Version: 0.06



Contents

0.1 Introduction

Lately there has been some discussion about a new unicode type, mostly due to a request from the Lazarus team to support unicode in file operations (for filenames, not handling of unicode files). A few proposals were made on the fpc-pascal maillist, abd some discussion followed, but it died out, and there a lot of details of the proposals were only discussed on subthreads.

I decided to try to summarize all positions and requirements, at least as I saw them as a kind of a discussion document. During the discussions I also detailed the requirements I had in mind a bit more, so I decided to write them down too.

Versioning:

- First version mostly my own writeup. Was originally meant to highlight the flaws that I saw in Florian's original proposal. There might still be some of the negative sentiment left, please skip it.
- Second version mostly Florian's feedback which I commented on
- Third version vastly expanded the Tiburón paragraph when CG lifted the veil a bit late July/early August, and the hybrid model.
- Fourth version added the "economics" paragraph, expanded the hybrid model and mentions Yury's proposal and wiki page.
- Fifth version(sept 2010) is mostly due to the new requirements now that FPC support compatible to Delphi/Unicode is becoming a possibility (cp-newstr). Some other details (OEMSTRING) were also added.
- Sixth (0.6) (dec 2010) version adds some minor completion of unfinished sentences and other minor clarifications

0.1.1 Tiburón

Tiburón is the codename for what is supposed to be the next version of Delphi (version 2008?), and is supposed to have unicode. While we currently do not follow Delphi compatability slavishly, it should only be broken if there are good reasons. A main reason for this is to not make life too hard on Delphi open source projects that also want to support FPC/Lazarus. Slowly details about Tiburón are starting to appear in CG oriented blogs. (e.g. Andreas Bauer's)

- A new utf-16 ref counted unicode stringtype is added.
  - s[x] doesn't take care of surrogates.
  - It is not yet clear if and how it supports endianness.
- Ansistring becomes a basetype for all 1 byte based encodings (ansi, codepages,UTF-8), based on the fact that for internal windows functions, UTF-8 is treated as a codepage.

- To define a stringtype for a certain (Windows) codepage enumeration value, type mycodepagestring = type ansistring (1251);
- Conversions that write a non UTF-8 codepage can be lossy.
- UTF-8 is codepage 65001 (ident CP_UTF8)
- codepage $FFFF is used for an "Rawbyte" ansistring that is never converted, it's binary copied into the target.
- probably value codepage $0 is used for the old ansistring. The conversions to and from this type (which codepage?) are not clear.
- It seems that the typing of ansistring has become stronger, and honor TYPE (as in something = TYPE ansistring) is now really an incompatible type.
- Conversions are done over UTF-16, but this might be a Windows implementation detail. (IOW on Unix use UTF-8)
- Windows has a separate codepage (OEM) for console. So in fact there are three encodings (OEM, Ansi and UTF16) in a Windows enviroment. There is a separate tag (CP_OEMCP) for the default OEM page. I suggest we predefine OEMSTRING = ansistring(CP_OEMCP);

This quick summary has four aspects I don't like for porting to FPC:

1. The use of windows specific codepage enumeration values in language syntax. However maybe they are really serious about the constants use, and this is livable. In my opinion it is the VCLs job to encapsulate the winapi gore, and if it can't be avoided, at least encourage a clean use. Daniel notes that there are not much platform independant choices to begin with.
2. The fact that conversions between codepages are automated and can fail. (see also the discussion about codepages in the critique of Florian's proposal) This means that if you use codepage strings, you must be very careful with your codepaths, so that you can be pretty sure that there aren't alternate paths that mutilate data.
3. UTF-8 and UTF-16 are scattered over two different types. This solution is non-orthogonal.
4. The big one, the compatibility break between Delphi2007- and 2009+. FPC can avoid this on a compiler level the same way they fixed the string=shortstring to string=ansistring move, but the library level is more difficult. D2007- expects e.g. Windows API functions to call the -A versions, while 2009+ calls the -W versions

They probably had the same as we problem for multiple-encodings types (see "Granularity of []") , but chose to keep this compiletime by dividing the types according to 1 or 2 byte granularity. Maybe this also has some advantages in the compiler (being able to treat tunicodestring and twidestring the same here and there). And they don't support UTF-32, probably because windows doesn't (or it isn't used)

Another question mark is the fact that a lot of new ansistring variants are introduced that are apparantly type safe. The question begs what stringtype is in e.g. variant (my guess: all non ansi ansistrings are converted to either widestring or a new tunicodestring field)

0.1.2 The encodings

The three main encodings are UTF-8, UTF-16 and UTF-32. An important property of these is that they are basically different ways to describe the same, so they can be convererted to eachother pretty easily and safely. Note that the multi byte encodings (16 and 32 (?)) also have big endian and little endian variants.

However for now I'm going to **forget the big endian and little endianess**. This kind of cross-platform compability is fairly rarely a problem. Only files that share that between different architectures need to insert conversions, and this can be better done manually. The same goes for arbitrary other sources that might have a different encoding. The difference is also only important on the perimeter of the system (when you load external data), since the system will mostly be in the same endianess

Besides these three main encodings, conversions of the string type to and from the older codepages could be useful too, because the world won't become unicode instantly, and ansistrings are here to stay for a while. Most notably Florian's proposal has some (potential) support for other codepages too, though not many details.

0.1.3 Economics of the encodings

In one of the unicode discussions Daniel posted this link: http://unicode.org/notes/tn12/ `http://unicode.org/notes/tn12/`. I just had some discussion about this in a different maillist on the subject of which is the ideal encoding, and here is my opinion some comments about encoding enconomics. Note that not all points are meant as arguments in favour of UTF-8 per se, just observations.

- First and for all, the question is mostly irrelevant since the choice of primary encoding (and endianness if>8) for a platform/target has been made already by the OS and the general ABI. Deviating from this to simply possible multiplatform programmers at the expensive of people programming for the platform natively is IMHO not an option.
- An often misinformed statement is that everything but ansi is worse in utf-8. This is not true, everything up from ascii to codepoint $0800 is equal in size between UTF-8 and UTF-16. This plane contains Cyrillic as well as several popular languages from the Semetic group like Hebrew and Arabic.
- The simplicity of UTF-16 is quoted in a lot of place, the above link inclusive. While some may see it acceptable to cut corners in applications, it is IMHO not acceptable to break full unicode compliance in a serious library, and most of all, a RTL. This means that most speeddependant routines in an app must be able to handle UTF-16 surrogates and maybe also endianness. I personally think that serious applications shouldn't cut corners either. Note though that surrogates don't hinder all string routines.
- Routines that don't need to process UTF-8 surrogates and encounter mostly Latin scripts are faster in UTF-8. (less bytes to move)

Btw I use http://www.unicode.org/roadmaps/bmp/ `http://www.unicode.org/roadmaps/bmp/` to quickly see what language groups are where in the BMP.

0.1.4 Granularity of []

One of the benefits of the discussion was that it called some attention to the s[] operator. First because it was a possible weakness of Florian's proposal (that got remedied later), but the more important one from a design perspective is what c:=s[5]; is supposed to mean with (s in [UTF8,UTF16,UTF32]).

Let's take utf16 for a moment, and assume we have 10 codepoints, and every second is a surrogate. Then there are three possible meanings:

0.1.4.1 Meaning 1: index means codepoints

In this meaning, a string is (a view on) an array of codepoints. So

c:=s[5]; means the 5th codepoint. A codepoint can be >2 bytes, so type of "c" must be able to contain a 32-bit value. The first 5 codepoints have two with surrogates so the address of the first char is @s[1]+5*2 + 2*2=@s[1]+14 (all in bytes)

Writing a character (s[5]:=c) is an even worse problem, since a codepoint written might not have the same size as the codepoint current;y at the at s[5], needing costly (O(n)) insertion routines.

0.1.4.2 Meaning II: index means granularity of the encoding

In this meaning the string is (a view on) an array with the ganularity of the encoding. So 1 in the case of UTF-8, 2 in the case of UTF-16 etc.

c:=s[5]; in UTF-16 means s[1]+5*2 =@s[1]+10 (all in bytes)

Writing a character (s[5]:=c) pretty much remains the same everything has the granularity of the encoding.

0.1.4.3 Meaning III: index means character

This is nasty, even UTF32 has the granularity of a codepoint. However printable characters may be composed out of multiple codepoints. This basically means the end of "char" as a separate type. Everything is a variable length string, and basic string operations have to be code on a lower level.

0.1.4.4 Granularity conclusion

(Note that the same problem also goes for Length(s). codepoints or elements in the granularity of the encoding?)

The problem with the array of codepoints is that typical code like

```
for i:=1 to length(s) do
  s[i]:=' ';
```

is very expensive since

- the address of s[x] depends on all codepoints before codepoints x. This can make the above loop quadratic in the number of codepoints jumps (on average (n^2)/2). Most platforms also use a procedure to iterate over codepoints.
- each codepoint assignment can possibly be an insertion or deletion of bytes, since the assigned codepoint can be smaller or larger than the codepoint already in place.

IMHO this opens a can of worms where we don't want to go[1]. However it might be an argument to (also) support UTF-32, since that does allow fairly easy char manipulation, with minimal limitations: If it is a routine that is not really much used, the simplest way to convert would be to do something like

```
procedure dosomething (var s:utf16string);
var internals: utf32string;
begin
  internals:=s; // force conversion to utf32.
  <<insert old ansistring code here, but only update "char" to a 32-bits type>>
  s:=internals; // convert back.
end;
```

Of course this is not perfect (e.g. charsets won't work because even a charset for the defined codepoints would be in the magnitude of 125k), but it is easy, and avoids messing too much with working code.

To state the obvious: to go there, we would have to forgo using the basic string types in standard routines, and code every reusable string routine on an assembler or pointer level.

0.2 Requirements

The requirements are a bit of a problem because there are several factors that are not compatible to each other (e.g. speed and ease of use), and tradeoffs vary. Anyway the main requirements in a very broad definition are:

- Ease of use
- Reasonable to good performance
- Compatible with normal ansistring handling as much as reasonably possible.

[1] Since we don't have any optimizations that optimize loops in an advance way, I don't think it is acceptable to waive this point in the hope that future optimizations will solve this.

- Compatible with Tiburón
- Multi platform aspects.
- Respect certain FPC traditions, most notably
  - the need to combine code from different origins/styles into one program. (e.g shortstring TP and ansistring Delphi code) code are currently combinable in one program, and a single directive controls the meaning of the "string" type to make it compatible on a per unit basis with both)
  - the fact that the entire RTL is mostly implemented in (FPC's) Pascal.
  - FPC being portable means nativeness on every platform. Not carrying conventions from other operating systems to operating systems where they are alien (e.g. POSIX on Windows)

Note: Most of the unices, but not all, use UTF-8, Windows use UTF-16.

0.2.1 Required "new" functions.

1. Regardless which choice is made for the default (see ?? on page ??), Length(s) should be available in both meanings: length in codepoints and in granularity length.
2. charat(n) - returns codepoint [n]... assuming we chose the encoding granularity.
3. charnext (strnext out of delphi compat?)

How much of these will/should be (partially) inlinable? Is it worth it? It seems that most libc's use functions, not macro's, which might be an indicator that procedural overhead is less than the actual operation.

0.2.2 The Windows "W" problem

Sideways related is the windows problem that on NT special functions must be called for unicode strings, all these functions end on -W instead of -A. Also all these symbols (and their record definitions) are typically organized in the windows header source as

```
{$ifdef unicode}
procedure xxx; (arguments);stdcall; external 'kernel32.dll' name 'xxxW';
{$else}
procedure xxx; (arguments);stdcall; external 'kernel32.dll' name 'xxxA';
{$endif}
```

The actual problem is that these calls don't exist (or work) on windows 9x. There are several solutions for this problem:

1. A combination of runtime OS detection and loading. Problem is that the windows header sets are huge, and there is a great potential for error.

2. Splitting the win32 target over unicode support.. So the current implementation is parameterized and move to a shared dir, and win9x target sets some types and defines, and imports these includefiles, as well as the NT-unicode target that defines UNICODE.

Personally I like the splitting. Note that the "win9x" target will still work on win NT/2k/XP, and is in fact a "real" win32. Note that the target names were picked in a hurry, maybe "win32" and "winnt" are better target names.

NOTE: I haven't seen any clear evidence yet to which set of API functions UTF_8 needs to be pasted (NT Only). If any.

0.3 The proposals

In the maillist discussion there were 3 proposals that I'll summarize shortly below.

0.3.1 Felipe's proposal.

Felipe's proposal was the first, and was mostly still oriented towards the direct File I/O problem. He proposed to use UTF-16 exclusively. Period.

Advantages

1. Simplicity

2. Carries Delphi compatibility to the extreme, introducing Delphi/Unicode UTF16 assumption on all platforms. Even if UTF16 is not the native unicode encoding.

Disadvantages

1. No way to support UTF-8, this means that all dealing with UTF-8 (the main encoding on Unix) must be manual on p(ansi)char level or through careful use of ansistring workarounds, or face heavy repeated conversion penalties. This also means code must be written to pass a readonly unicode string to a library on unix, instead of simply passing pwidechar(s). It is a windows centric proposition

2. No utf-32, so also no simple way

Keep in mind that this also means some complications for e.g. standard file I/O, that must change from UTF-8 to UTF-16.

0.3.2 Marco's proposal

This proposal was more in line with earlier discussions on core, simply have three separate types for the three encodings, that autoconvert reasonably, and the implementation is nearly the same. To keep RTL size down, most system calls would only accept strings in the system encoding, except for VAR parameters that need to be wrapped or double implemented.

So for clarity: an utf8string, utf16string and a utf32string type.

Advantages

1. The string types that a routine use signal the encodings it accepts/returns.
2. Maximum speed for code that uses only one encoding, no conversion, no runtime behaviour.
3. The fact that the types have exactly the same content in a different representation (4 types, together with UTF-32 and the COM widestring) made me hope that the implementation would not be that much more complicated than one + a bunch of special options and directives.
4. Interfacing with systems with a different encoding is simple. Convert to correct type if not already, and then typecast.
5. Tiburón code could simply use UTF16 string everywhere (a simple {$H like directive), and be very to totally compatible, and yet mixable.

Disadvantage

1. Most new types, thus also the most conversions.
2. Separate types, so one can't pass UTF-8 string to a procedure with a var or out parameter of UTF-16 type.
3. Only overloading and conversion as instrument for routines that must accept multiple encodings. Not unlike ansistring and shortstring IOW with the same problems.
4. More types also means a lot more vt<x> constants in tvarrecs, variants etc.
5. Prefix records of types can't be Tiburón compatible

0.3.2.1 Aliases

To make this work properly, there will be some additional aliases:

- An alias to a type that always is the same as the system encoding. If you use this you are always safe performance wise.
- An alias to utf16string of whatever identifier Tiburón uses for

This also means that encoding agnostic code should use the system encoding, since the average string will be probably in the system encoding

0.3.3 Florian's proposal.

Florian proposed to have a single unicode type that can represent the three encodings (UTF-x), and maybe others too (the old ascii codepages as well as LE vs BE). The principle is the same as ansistring, additional needed info is prefixed at addresses before s[1]. Currently it is only the encoding type, but it could be expanded.

There are a lot more implementation details to be resolved in this proposal.

Florian says the following about the granularity of the type.

> to overcome the indexing problem efficiently when using an encoding field (this is not about surrogates), we could do the following: introduce a compiler switch {$unicodestringindex default,byte,word,dword}. In default mode the compiler gets a shifting value from the encoding field (this is 4 bytes anyways and could be split into 1 byte shifting, 2 bytes encoding, 1 bytes reserved). In the other modes the compiler uses the given size when indexing. For example, a Tiberion (or how is it called?) switch could set this to word.

Later however he says (in response to the below granularity challenge)

> I described this already in detail in my first mail: just in one of the four bytes available for storing the encoding.

Now I'm confused :)

Anyway about the performance he says:

> The approach has the big advantage, that you really need all procedures only once if desired. For example e.g. linux would get only utf-8 routines by default, utf-16 is converted to utf-8 at the entry of the helper procedures if needed. Usually, no conversion would be necessary because you see seldomly utf-16 in linux applications so only the check if the input strings are really utf-8 is necessary, this is very cheap because the data is anyways already in a cache line.

He also says

> Keep in mind in your response, that we want also handle other formats than utf-8 or utf-16 if needed :)

Michael says:

> For the LCL/fpGUI/MSEGui programmers, nothing changes, > you can even throw away your own conversion routines. > You need only a single call just prior to passing a string > to the OS/GUI system: ForceEncoding(). No ifdefs needed, > all is transparant.

The type is a bit too complex to have a series of simple advantages and disadvantages, so I just going to describe some of the problems, and ask for clarification.

0.3.3.1 The biggest problem: can't declare what type to expect.

My initial reaction was "oh my, a runtime type in Pascal, what about performance? It will be pretty much likes variants, and they are known to be slow. We will become Perl/Python"

However while I still have serious doubts about performance, that's not the bigger problem. Since with pretty much any solution you can always isolate the speed dependant part, force the encoding to be constant (preferably the system encoding), and be done with it. Moreover, there is much to say for having only one string type, even if it is polymorphic internally.

The *bigger* problem however is that you don't declare the type of the encoding anymore in parameters, local variables and return type. This means manual insertion of Michael's Enforceencoding calls everywhere, also in existing Tiburón code. It invalidates my own (but agreed: not Florian's requirements) that existing code remains running with only some global mode settings. (assuming Tiburón is "existing code"). Or, generalized: if you synthesize an application using code from various sources you'll need

I can illustrate that with two examples or thought experiments:

0.3.3.2 Existing code

Assume I have a unit with UTF-16 Tiburón code. And and some unit with UTF-8 code of Lazarus descent where I globally replaced "ansistring" by unicodestring (or whatever identifier for the native type) to upgrade it to "native" unicode on an Unix target.

Now we want these to work call eachother, and neither of these is prepared for the polymorphic type to contain the wrong encoding. Worse, literals in the Tiburón code will probably be created in the native (UTF-8) encoding. In turn, the UTF-8 routines might receive occasionally a string that has passed the Tiburón code and contains code that assumes UTF-16 encoding. The only solution is to audit the _entire_ source code for all these points, and insert ForceEncodings() statements for all parameters and after assignment of a literal. Here another potential problem surfaces, an empty string might not be forcable.

This is an extremely hard sell to Delphi users, and IMHO not necessary anyway. Something will have to be done about this. A solution would be the hybrid proposal, see the separate paragraph further down. It is more or less the declarative behaviour of my proposal combined with the implementation of Florian's.

0.3.3.3 The granularity

The problem with the granularity lies a bit in the same region as the last: if you have a procedure you must be prepared to handle all types. Now assume I honour that, and I am trying to make a procedure that understands both encodings, e.g. a dual encoding version of the "granularity" problem above. Then according to Florian's first quote above *I only have one compiletime granularity while the type of my unicodestring is defined runtime !*

```
{$unicodestringindex <what to put here?>}
procedure myuniversalstringroutine(s:tunicodestring);
begin
   if encodingof(s)=utf_8 Then
     begin
       for i:=1 to length(s) do  // s in single bytes
         s[i]:='a';              // s[i] in single byte values. type of literal?
     end
    else
     begin // utf 16
       for i:=1 to length(s) do  // length(s) in 2 byte values
         s[i]:='a';   // s[i] in two byte values. type of literal?
     end
end;
begin
  myunversialstringroutine(getutf16stringroutine);
  myunversialstringroutine(getutf8stringroutine);
end;
```

The conclusion of this is IMHO that shift size should be part of the runtime string too, iow a value of 1,2,4 somewhere at negative offset of the pointer. This is a performance penalty, since s[4] is then a more runtime construct.

0.3.3.4 Performance

A runtime solution is always slower as a compiletime one. While performance isn't my biggest gripe, the problem is that I only see a small advantage in return: working VAR parameters and a lower need for overloading. For that we see a lot more checks done (because the encoding check must be after the nil check which will complicate codegeneration).

Florian claims to partially earn this back with less conversions in all, but I don't buy that, except maybe in cases like (mostly GUI) apps with QT on *nix (where widgetset (UTF16) and system encoding (UTF8) are different). Simply having an type-alias for whatever encoding is the system encoding will achieve the same. Moreover, the decision which type to convert lies with the compiler which has generally more information at its disposal than the runtime library. Take for instance the following example:

```
var s1: utf8string;   // utf-8 is the system encoding, we're on unix
    s2: utf16string;
s1:=someinit8();
s2:=someinit16();
s1:=s2+s1;
utf8routine(s1);
```

(note that for Florian's example, all string types are the same, in his case, read the declarations of s1 and s2 as "strings initialised filled with a utf-8/16 value)

Now the runtime libs can probably not exploit the fact that the system encoding is more useful, and s1:=s2+s1; might end up converting the utf-8 type to utf-16, and storing the utf-16 result in s1. And then the check in utf8routine() will have to change the encoding again.

Also the "leaf out routines" argument is IMHO bogus, since if the types of my proposal autoconvert (not unlike uniquestring()), the more complex routines like the bulky floating point and datetimeformatters could also be available only in the system encoding (which is most likely to happen), give or take a few small wrappers to work around VAR parameter problems.

0.3.3.5 Alternate encodings.

(this paragraph is academic since we need to support other encodings because of Delphi, it was written before this was known though)

Florian also mentioned an interest in supporting the old codepages as part of the requirements. I don't know if that was only a teaser because his proposal had more leeway for that or because he *really* saw a case and a need for that.

However while I entertained the idea as interesting for a while, I'm not so convinced this is doable for two main reasons,

- the UTF-x to UTF-y conversions are guaranteed to work if not corrupt, and if there are corner cases, they are far and few. But the codepages only accept a real small set of the possible codepoint set of the UTF-encodings and also eachother. The errorhandling is IMHO a problem.
- Because the type of the polymorphic doesn't change unless forced, these strange encodings could penetrate everwhere in your codebase when simply strings are passed on unmodified. The amount of exceptions of unexpected encodings, and conversion failures all over your (till now working) code is confusing, unless you want to manually try except all string code in case some conversion goes wrong.

0.3.3.6 Florian's response

The discussion about this article doesn't seem to have changed much about each parties viewpoint. Except maybe the "existing code" problem,[2]

(quote Florian)

Indeed, it requires some work but there are several possibilities:

1. add a switch for runtime checks about string encoding
2. add a switch to enforce encoding at procedure entries and for function results

[2]Note that existing code is not only code that is "old" or "Tiburón" but in general all code that can only accept one encoding.

The code needs to be reworked anyways.

(...end quote..)

I think this is butt ugly, and overly complicated, but at least it fixes my most major problem. Maybe if we can predeclare a lot of these as types, we can actually confine the clutter.

note: see also the 0.3.5.1 on page 17 paragraph, and the hybrid paragraph in general. There are complications.

0.3.3.7 The good points

In some ways this proposal was better than the Windows centric Tiburon implementation, in the sense that it unifies UTF8 and UTf16 in one stringtype. Apparantly Codegear even put more stress on cheapness of [] than I did. In theory a stringtype with a granularity field (in the TAnsiRec), could host both UTF8 and UTF16. This might cause Codegear pain when going multiplatform. If they persist in UTF16. Of course we'll never know for sure if this was a lost chance or not. This is likely, since for them multiplatform is mostly only a "feature" in addition to the core win32/64 product.

0.3.4 Yury's proposal

Yury wrote something up independantly at FPC wiki about FPC Unicode support `http://wiki.freepascal.org/FPC_Unicode_support`. It is the same basic idea as Florian's: encodingtype and granularity-of-encoding in the prefix of the string. He goes a step further and also seems to hint on reimplementing existing types on this scheme. (which is not realistic for shortstring, and maybe widestring).

What I like in Yury's proposal is that he combines the implementation from Florian with the declaration that shows real types that I favour, in short, essentially it is the hybrid detail of the next paragraph in the rough. The hybrid model does divide some of the types over two types, the new unicodestring and ansistring (the codepage stuff, if we do that, there is no need to be Tiburón incompatible)

0.3.5 The hybrid model

This is just a short thought experiment, this part hasn't been discussed with Florian and Michael much yet (though Yuri seems to come up with it independantly). The main reason is that the typing is my main grudge against Florian's proposal, and the performance less. It builds a bit on Florian's willingness to tackle some of those with directives. If that gives enough leeway to define types, Florian's proposal morphs into this hybrid model.

So assume we combine Florian's and some of the requirements (but not implementation) that are the basis for Marco's example. This means one base unicode

type that can be parameterized to four types for declaration purposes (a single implementation of generic runtime dependant unicodestring as per Florian's proposal, but separate (sub)types per encoding (TUtf8string,TUtfstring16 and TUtfstring32)). These latter might be not real (compiler) types, but defined like below.

```
Type
  tutf8string = type tunicodestring(Mandatory_UTF8); // or however we style the modif
alternate syntax (?), more in Florian's style with directives
type
  {$unicodetype mandatory_utf8}
  tutf8string = tunicodestring;
  {$unicodetype general}
```

However because these forced types are 100% compatible with the full type, there is less of a multitude of overloads for VAR or overloading of helpers (for e.g. variant which only contains the general type).

- the desired compiletime declarative behaviour, to be able to declare when a certain routine only accepts/expects a certain encoding.
- the ability to have compiletime type knowledge to rearrange expressions to prefer a certain encoding result (see the performance paragraph) by using a different declaration (much like the Tiburón ansistring), if all components are typed.
- In Tiburón mode, the string type is equal to TUTF16string, but can be mixed with any of the other types.
- On implementation level, a single runtime implementation. No 3 ways of overloading.
- The whole situation is then a bit analog to shortstring vs shortstring[] (from a typing point of view). All RTL routines are var shortstring, and accept all. However if you want to only support a certain size (like extensions), you can declare it using var s:shortstring[2]. But the unicode equivalent would be expected encoding, not size. Also e.g. variant would hold an FPC unicodestring, which is compatible without conversion to utf8string, utf16string,utf32string

The main advantage of would be keeping the number of type dependant (not the more general routines) down, but to be able to retain the compiletime typed behaviour. Slowly I'm convinced this might be a doable way, but I need Florian's input for that.

As a bonus, expanding this hybrid with Tiburón functionality is also possible, with quite high Tiburón compat, at the expense of having two UTF-8 types:

- Implement the hybrid type as above. Only TUnicodestring only has the base three encodings.

- Implement the Tiburón ansistring. utf-8 inclusive. This also includes the codepage support then.
- In Delphi (Tiburón?) mode, the default unicodestring is an alias for TFlorianString(talwaysutf16).

This trick allows to simply add Tiburón code under the relative IFDEFS, and keep it working. And to gain maximum performance (avoid too much conversions in one codepage) on Unix utf-8 people would could remove the Tiburón flags on a per unit basis after inspecting the encoding state of an unit.

The problems that I can think of, is that there is still a VAR problem, and the type and conversion situation in the compiler might get complicated (a lot of combinations), even though the number of overloads might be less.

0.3.5.1 Problems of hybrid: var

- VAR remains a problem, but afaik it is fixable.

Assume we have RTL routine that does

```
procedure stringroutine (var s:TUNICODESTRING);
begin
  forceencoding(s,utf16); // code only can deal with utf16
  process;                // the utf16 processing code.
end;
and
var n : tUTF8String;
begin
   {assign n}
  stringroutine(n); // we can pass, since this is not a fully different type, but a T
  // BUT: here n would be UTF16, a violation of the type declaration.
end
```

This means that the compiler should insert a forceencoding after passing a string with encoding affinity to a generic VAR parameter. I hope that is doable.

- As in Florian's original proposal the [] operator gets more expensive in generic routines. In non generic (roughly equivalent to Rawbytestring in Tiburón terms, but then also for UTF16) it is less of a problem, since the typing fixates the granularity? This could offer a nice solution in the sense that high speed routines could be overloaded with typed equivalents for speed, while not so interesting routines could rely on the "general" type with runtime granularity.