A Proposal to Free Pascal for true Unicode Character and String Types

Unicode Character String handling is a question that keeps coming up on the Free Pascal Mailing lists and, empirically, it is hard to avoid the conclusion that there is something wrong with the way these character string types are handled. Otherwise, why does this issue keep arising?

One answer might be that “Unicode” is an overloaded term and due to Microsoft calling their early implementation of UTF-16 – Unicode – there is confusion over what is Unicode, compounded by the type AnsiString (whose name implies legacy code pages) also covering UTF-8. Further confusion exists with type names that reflect their implementation from Unicode's original synonymous use with UCS-2 and with no update to reflect the evolution of UCS-2 into UTF-16. However, it is probably not that simple.

Supporters of the current implementation point to the rich set of functions available to handle both UTF-8 and UTF-16 in addition to legacy ANSI code pages. That is true – but it may be that it is also the problem. The programmer is too often forced to be aware of how strings are encoded and must make a choice as to which is the preferred character encoding for their program. There then follows confusion over how to make that choice. Is Delphi compatibility the goal? What Languages must I support? If I want platform independence which is the best encoding? Which encoding gives the best performance for my algorithm? And so on.

Another problem is that there is no character type for a Unicode Character. The built-in type “WideChar” is only two bytes and cannot hold a UTF-16 code point comprising two surrogate pairs. There is no char type for a UTF-8 character and, while UCS4Char exists, the Lazarus UTF-8 utilities use “cardinal” as the type for a code point (not exactly strong typing).

It all used to be much simpler. In Jensen and Wirth's original Pascal specification, the “char” type was defined such that “a value of type char is an element of a finite and ordered set of characters”. The specification then went on to note that there was no single standard for character encoding (ASCII and EDCDIC were probably what they had in mind) and thus a character's encoding and collation were implementation dependent. Strings were just “packed array [1..n] of char” with the managed string type coming later. There was no choice: there were just characters and character strings encoded in a platform specific manner.

In order to stop all this confusion I believe that there has to be a return to these fundamental concepts. That is the value of a character type represents a character, while the encoding of the character is platform dependent and a choice the compiler makes and not the programmer. Likewise a character string is an array of characters that can be indexed by character number, from which substrings can be selected and compared with other strings according to the locale and the unicode standard collating sequence. Let the programmer worry about the algorithm and the compiler worry about the best implementation.

The Proposal

I want to propose a new character type called “UniChar” - short for Unicode Character, along with a new string type “UniString” and a new collection “TUniStrings”.

The motivation for this is to provide a proper foundation for the handling of Unicode Characters and Character Strings in FPC. Pascal character and string types have been subject to an evolution over the years that has been driven by Delphi and which has added different string types, often without proper consideration for consistency. Too often the motivation in Delphi appears to have been to quickly support a Microsoft Windows API or convention without necessarily worrying about maintaining a consistent model for handling character strings.

It is my assertion that in order to restore consistency, there needs to be a fundamental type for a Unicode Character that is independent of how it is encoded on a given platform.

Improved security is also a motivation for this proposal. Some attacks depend upon poor handling of extended encodings, such as in UTF-16, and a better model for handling Unicode character strings, with more opportunities for common code, could be a very useful countermeasure.

It is hoped that one day, UniChar, UniString and TUniStrings, will become synonyms for char, string and TStrings.

Design Rationale

The intent is to create a character and string handling design that is natural to use with the programmer rarely if ever having to think about the character or string encoding. They are dealing with Unicode Characters and strings of Unicode Characters and that is all. When necessary, transliteration happens naturally and as a consequence of string concatenation, input/output, or in the rare cases when performance demands a specific character encoding.

There is also a strong desire to avoid creating more choice and hence more confusion. The intent is to “embrace and replace”. Both AnsiString and UnicodeString should be seen as subsets or special cases of the proposed UniString, and with concrete types such as AnsiChar, WideChar and WideString, other than for legacy reasons, existing primarily to define external interfaces. Such types should only be needed at the periphery and most programmers need not be aware that they exist. Similarly, complex string handling libraries (e.g. LazUTF8) should no longer be required for other than legacy reasons.

The proposed change is grounded on the “UniChar” type.

UniChar

A “UniChar” is a container for a single Unicode Character.

It is proposed that the encoding method for the character is left undefined and a local implementation matter. In practice, the most likely encoding may well be UCS-4, and four bytes will be required per character (which is the same for any UTF). However, the intent is to permit local optimisations whenever possible as well as to give a clear indication to programmers that they should not rely on the binary values of a UniChar and, instead, should always refer to them as typographical symbols or use the Unicode standard representation for a literal character as a code point (e.g. U+0041 is the code point for an upper case Latin 'A').

The name “UniChar” is used instead of the perhaps more obvious “UnicodeChar” because UniCodeChar already exists and is a synonym for WideChar. Furthermore, Unicode in common use is often used as a synonym for UCS-2. The existing definition of UnicodeChar comes from this common usage; it is too limited a type for general use.

In practice, a function may exist to determine the character encoding of a UniChar at run time, but this does not detract from the fundamental idea that a UniChar is a container for a Unicode Character and is semantically the same regards of how it is encoded on a given platform.

UniString

A “UniString” is defined as a reference counted container[1] for an array of UniChar, where each element of the array is constrained to use the same encoding method. The encoding method used for a given “UniString” is determined dynamically and can be any well known encoding method, including Unicode Transformation Formats (e.g. UTF-8, UTF-16, UCS-2, UTF-32, etc.), ASCII and ANSI code pages.

The encoding method used for a given UniString can be determined by a standard function and can be changed, invoking transliteration from one encoding method to another. However, most programmers need not be aware of the actual encoding method.

A UniString is a more “concrete” type than the abstract UniChar, given that the encoding method is part of the type and is potential visible and modifiable by the programmer. This is because UniStrings will often be used with external devices (that have specific encoding requirements) or in algorithms where the encoding method can impact on performance. Indeed, the idea is that while it may be necessary to transliterate from one encoding to another when strings are compared or written to an external device, transliteration should only occur when necessary and should be avoided otherwise. Thus, for example, a string read from a UTF-16 source should remain encoded in UTF-16 until it needs to be written to a UTF-8 sink, if at all.

Similarly, if the encoding method impacts performance for a given algorithm, then the programmer has the option to force transliteration of input strings that are not in the ideal encoding.

However, in most cases, the programmer should not have to be aware of the encoding method used for a given UniString.

The above definition also means that a typical implementation of a UniString has a very similar storage requirement to a current AnsiString (cp_utf8), occupying up to four times the character width of the string plus two bytes to identify the encoding method. Some implementations may also want to add further information. For example, to indicate a fixed width encoding method (one, two or four bytes) and hence allow for rapid indexing.

The Unicode Byte Order Mark (BOM), used in multibyte UTF formats (e.g. UTF-16) to indicate the byte order for each code unit, should not be used in a UniString; the byte order should be implicit in the encoding method specified for the UniString.

A UniString may not be used as a container for general binary data. There is no encoding method implying “no encoding” and any attempt to use a UniString for binary data may result in data corruption or an invalid Unicode code point exception being raised. The implication is that a separate type such as TBytes should be used for binary data.

A UniString can have an “undefined” value as is the case for an uninitialised UniString. An exception is raised if an attempt is made to reference the value of a UniString with an undefined value.

1 Similar to the existing AnsiString and UnicodeString types.

TUniStrings

A “TUniStrings” is defined as a collection of UniStrings, where each member of the collection is constrained to use the same encoding method. It is very similar to the existing TStrings. A typical implementation will follow the TStrings model but additionally adding two bytes to the storage requirement to identify the encoding method.

By forcing all strings in a TUniStrings to have the same encoding method, the implementation is considerably simplified.

Code Pages

The existing TSystemCodePage type is carried forward to UniStrings, and includes the following codepages:

```
cp_utf8    = 65001; {Unicode UTF-8}
cp_utf16BMP = 1200; {Unicode UTF-16, little endian byte order
                    (restricted to BMP of ISO 10646 – delphi compatible)}
cp_utf16BE = 1201; {Unicode UTF-16, big endian byte order}
cp_utf16LE = 1202; {Unicode UTF-16, little endian byte order}
cp_ucs2    = cp_utf16BMP; {Unicode UCS-2}
cp_utf32LE = 12000; {Unicode UTF-32, little endian byte order}
cp_utf32BE = 12001; {Unicode UTF-32, big endian byte order}
```

In addition, cp_utf16 and cp_utf32 will also be defined as synonyms for the variant of UTF-16 or UTF-32 that is the default for the compilation environment. e.g. For Windows:

```
const cp_utf16 = cp_utf16BMP;
```

The cp_none code page identifier is not valid for a UniString.

The TSystemCodePage for a UniString may be changed or identified by the following standard procedures:

```
procedure SetCodePage(var s: UniString; CodePage: TSystemCodePage);
```

This function is used to set the codepage (encoding method) used for a UniString. If the input UniString is undefined then the output UniString is an empty string with the specified codepage.

Note that transliteration always takes place from the current encoding method to the new one if the code page is changed. There is deliberately no option to change the encoding without transliteration.

The following function may be used to return the current code page:

```
function StringCodePage(const s: UniString): TSystemCodePage;
```

This function returns the code page identifier for the encoding method used for a UniString.

Note: an uninitialised UniString has an undefined code page. The StringCodePage function may raise an exception when used on an uninitialised UniString variable.

Literals

Character and string literals may be defined using typographical symbols or, on a per character basis, as a Unicode code point number.

- Character and string literals defined using typographical symbols are always enclosed in single quotes e.g. 'a', or 'xyz'.
- Individual characters defined as a Unicode code point number are given in the format:

    #U+<hexadecimal number>

    For example: #U+0041 is the Unicode code point number for an upper case letter A in the Latin alphabet.

Both types of string and character literals may be concatenated, with no intervening white space, to create a single string literal. For example:

```
#U+0041' string'
```

Is equivalent to 'A string'.

The encoding method used for string literals is platform and version dependent and cannot be assumed by portable code. For example, UTF-8 may be used for Linux, while UTF-16 may be used for Windows.

Legacy character literals in the format #<number> may still be used. They are interpreted as identifying a single byte ANSI character using the DefaultSystemCodePage. A literal defined using this format is an AnsiChar literal. It is possible to assign AnsiChar literals to UniChar variables and to concatenate them with UniString literals. The rules for assignment and concatenation given below apply in these cases.

System Global Variables

- **var** DefaultSystemCodePage: This is an existing global variable and determines the default ANSI code page. It is used to identify the encoding method used for the existing single byte “AnsiChar” type.
- **const** DefaultEncodingMethod: TSystemCodePage;

    This is always set to the encoding method that is the default for the compiler and version. For example. It is used for string literals. Typically, UTF-8 or UTF-16.

- **const** DefaultByteOrder: (big_endian, little_endian);

    This defines the default byte order used for UTF-16 and UTF-32 code units.

- **var** DefaultCollationSequence: TCollationSequence;

    This gives the default Unicode Collation Sequence used for UniChar and UniString comparison (see below).

- **var** DefaultStreamCodePage: TSystemCodePage;

  This is the default code page for streams and is platform specific. It is typically set to UTF-8.

String Initialisation

A string is usually initialised by assigning to it another string, a string literal or the result or output of a function or procedure. However, a UniString may also be initialised from raw data.

Two new standard procedures are defined:

```
procedure SetString(out s: UniString; codepage: TSystemCodePage;
                                          buf: PAnsiChar; len: integer = 0);
procedure SetString(out s: UniString; codepage: TSystemCodePage;
                                            buf: PWideChar; len: integer = 0);
```

These functions dispose of any current contents of “s” and replace them with a UniString encoded using the given codepage and copied from the string of length 'len' bytes given by buf. In both cases, “codepage” identifies the encoding of the source string. In the first case, this must be compatible with an AnsiString. In the latter case, this must be compatible with a WideString.

If the length is given as zero then the input string is assumed to be null terminated. Otherwise, the length is given explicitly by “len”.

String Length

When applied to a UniString, the standard function “length” returns the number of code points (Unicode Characters) in the string. This may differ from the byte length of the string.

When the UniString encoding method has a fixed element size (e.g. single byte ANSI, UCS-2 or UTF-32) then determination of the string length is fast, needing only to divide the byte count by the element size to find the length.

When the element size is variable (e.g. UTF-8 or UTF-16), the string must be scanned to count the number of code points. However, due to the self-synchronising nature of UTF-8 and UTF-16 encodings, the encoding does not have to be interpreted in order to achieve this.

- With UTF-8, it is only necessary to perform a byte scan looking for the start of each code point in order to count the code points found.

- With UTF-16, it is only necessary to scan each successive two byte pair, treating any surrogate pair found as a single code point.

With variable element size encodings, the option also exists for the compiler to speed up multiple length requests by caching the result the first time a string length is determined, invalidating the cached length only when the string changes.

Byte Length

A new standard function “bytelength” is also defined to return the byte length of the internal buffer holding the string (including null terminator) for cases where this may be useful (For an example see the section on External APIs).

```
Function bytelength(s: UniString): integer;
```

SetLength

The standard procedure “SetLength” is also defined for a UniString.

```
Procedure SetLength(var s: UniString; len: cardinal);
```

This sets or modifies the length of the UniString such that the underlying buffer may contain “len” Unicode characters in the UniString's codepage plus a trailing null character. This may result in string truncation if the length is less than the current string length (in characters). If the string length is increased then it is right padded with null characters. An exception is raised if the UniString is undefined.

Generally, the programmer cannot assume that a UniString's buffer can contain more bytes than are necessary to encode the string that it holds. The exception is after a call to SetLength when it will be large enough to contain the requested number of characters. However, this only remains valid until the string is next modified. For an example of when this is useful, see “External APIs” below.

Note: the above definitions mean that bytelength normally returns the encoded length of the string including a null terminator. The exception is after a call to SetLength which increases the buffer size, when it returns the number of bytes in the extended buffer. This allows it to be useful when passing the UniString's internal buffer size to external functions that store characters directly into the string's internal buffer.

Assignment Rules

It should be possible to assign all existing character types to a UniChar.

- The FPC manual defines the char type as containing one ASCII (read as AnsiChar) character and hence when assigned to a UniChar, the Unicode Character assigned to the UniChar will be set to the equivalent to the ANSI character associated with the single byte value of the char, encoded according to the default ANSI system code page.

- The FPC Wiki currently defines a WideChar as “exactly 2 bytes in size, and contains one Unicode code point (normally a character) in UTF-16 encoding”. This is a constrained UTF-16 with no UTF-16 surrogate pairs. When a WideChar is assigned to a UniChar, the Unicode Character assigned to the UniChar will be set to the Unicode Character encoded as the two byte UTF-16 value. The byte order assumed will be that given by the system global DefaultByteOrder for the WideChar[2].

- UCS2Char, UniCodeChar character types may also be assigned to a UniChar, with UCS2Char and UniCodeChar assumed to be a synonym for WideChar.

- UCS4Char is assumed to contain a UTF-32 encoded Unicode Character. The Unicode Character encoded as the value of the UCS4Char is assigned to the UniChar.

2 SetString, as defined above, will need to be used for UTF-16 code points split across two widechars.

- An integer may also be assigned to a UniChar. In this case, the integer is interpreted as the integer value of a code point to which the UniChar is then set. If the integer value does not correspond to a valid code point then an exception may be raised.

Assigning a UniChar to an older char type is potentially more complicated and may result in a transliteration error:

- When a UniChar is assigned to an AnsiChar, if the Unicode Character it contains may be encoded using the default ANSI system code page then the char is set to the the one byte value that encodes the Unicode Character in the default ANSI code page. Otherwise, an exception is raised.

- When a UniChar is assigned to a WideChar, UCS2Char or a UniCodeChar, if the Unicode Character it contains may be encoded using UTF-16 as a single two byte value then the WideChar, UCS2Char or UniCodeChar is set to this two byte value using the DefaultByteOrder. Otherwise, an exception is raised.

- When a UniChar is assigned to a UCS4Char, the UCS4Char is set to the UTF-32 encoding of the Unicode Character given by the UniChar.

- When a UniChar is assigned to an integer type (e.g. cardinal) then the resulting integer value is the integer value of the character's code point.

Other string types may also be assigned to a UniString.

- When an AnsiString is assigned to a UniString, the UniString is set to the same encoding method as used for the AnsiString and assigned the same value. An exception is raised if the AnsiString's code page is cp_none.

- When a WideString or UnicodeString is assigned to a UniString, the UniString is set to the UTF-16 encoding method with the DefaultByteOrder, and assigned the same value.

- When a shortstring is assigned to a UniString, the shortstring is assumed to contain an ANSI character string encoded according to the DefaultSystemCodePage. The UniString's encoding method is set to the DefaultSystemCodePage and the value of the shortstring copied to the UniString.

Note: assignment of a PAnsiChar or a PWideChar to a UniString is not supported. This is because the actual character encoding is ambiguous. SetString must be used to assigned a PAnsiChar or a PWideChar to a UniString.

A UniString may also be assigned to other string types.

- When a UniString is assigned to an AnsiString,

  - if the UniString's encoding method is supported by the AnsiString then the AnsiString is set to the same encoding method as the UniString and given the same value.

  - if the UniString's encoding method is not supported by the AnsiString (e.g. UTF-16) then an attempt is made to transliterate the value of the UniString to UTF-8. If successful then the AnsiString is set to the UTF-8 code page and given the transliterated value of the string. Otherwise an exception is raised.

- When a UniString is assigned to a UnicodeString or a WideString:
  - if the UniString's encoding method is UTF-16 in the DefaultByteOrder then the UnicodeString or WideString is given the same value. However, if the UniString contains any UTF-16 surrogate pairs, an exception is raised.
  - if the UniString's encoding method is not supported by a UnicodeString or a WideString (e.g. UTF-8) then an attempt is made to transliterate the value of the UniString to UTF-16 with no surrogate pairs. If successful then the UnicodeString or WideString is given the transliterated value of the string. Otherwise an exception is raised.
- When a UniString is assigned to a shortstring, if the contents of the UniString is less than 256 Unicode Characters and can be transliterated to the DefaultSystemCodePage, then the UniString is transliterated to the DefaultSystemCodePage, and the result copied to the shortstring. Otherwise an exception is raised.

Note that the above provides an assignment path between AnsiStrings and UnicodeStrings or WideStrings via a UniString. The compiler may chose to permit assignment between these string types by creating a temporary UniString as the intermediary through which the transliteration from one encoding method to the other is performed.

Concatenation

The operator '+' is used to perform string concatenation.

- If the encoding method used by the two strings is the same then the result is a UniString with the same encoding method and consisting of the encoded bytes of the first string followed by the encoded bytes of the second string.
- If the encoding method used by the two strings is different then the result is a UniString with an encoding method set to the DefaultEncodingMethod and comprising the concatenation of the two strings with one or both, if necessary, transliterated to the DefaultEncodingMethod.

The '+' operator also applies to UniChar operands. Functionally, a UniChar operand is expanded to a UniString prior to performing the concatenation.

If the '+' operator is used to concatenate a UniChar with a UniString then:

- If the UniChar can be encoded using the same encoding method as that assigned to the UniString then the resulting UniString has the same encoding method as the UniString operand, and comprising the concatenation of the UniChar and UniString in the order implied by the order of the operands.
- Otherwise, the UniString is transliterated to the DefaultEncodingMethod and then concatenated with the UniChar as above. The resulting UniString is assigned the DefaultEncodingMethod.

The above rules also apply to the built-in “Concat” function.

Coercions

The PUniChar type may be defined as:

```
type PUniChar = ^UniChar;
```

It cannot be used to coerce another character type to a UniChar and a compile time error should result if this is attempted. The reason for this is that the data storage structures may not be compatible and, anyway, the internal representation of a UniChar is undefined. Likewise, coercing a UniChar using PChar, PAnsiChar or PWideChar should also result in a compile time error.

Similarly

```
type PUniString = ^UniString;
```

As above, this is just a pointer to a UniString and cannot be used to coerce any other type to a UniString given that the data structures are unlikely to be compatible. However, the following coercions may be useful:

- A UniString may be coerced to a PAnsiChar or a PChar[3] if the encoding method matches any of those supported by an AnsiString (e.g. UTF-8). The result is a null terminated string using whatever encoding method is assigned to the UniString. An exception is raised if the encoding method is not AnsiString compatible.

- A UniString may be coerced to a PWideChar if the encoding method is UCS-2 or UTF-16. The result is a null terminated string using the UCS-2 or UTF-16 encoding. An exception is raised if the encoding method is not UCS-2 or UTF-16.

- A UniString may be coerced to a PByte. The result is a pointer to the buffer containing the encoded string (the bytelength function defined above may be used to determine the buffer size). This coercion is used when direct access to the encoded data is required. For example, when computing a message digest or when encrypting the string.

Wherever the result is unambiguous, implicit coercion should be supported. For example, when assigning a UniString to a PAnsiChar. However, transliteration should not be automatic as each pointer type (PAnsiChar and PWideChar) supports multiple codepages.

Note: The PAnsiChar, PWideChar or PByte value returned will be a pointer to the actual buffer used to contain the encoded value and is valid only while the original UniString remains in scope and is unmodified. Any use of the pointer to modify the contents of the buffer should only be made with great care if the UniString is not to be corrupted.

3 PChar should be avoided for Delphi Compatibility as it implies PWideChar in Delphi.

String Indexing

A UniString is a (one based) array of UniChars and hence may be indexed to obtain the UniChar in "i*th*" position e.g.

```
var
  line: UniString;
  c: UniChar;
begin
  c := line[3];
```

Where the value of the variable 'c' will be assigned the third Unicode Character (code point) in the string, regardless of the encoding method used for the character string.

When the UniString encoding method has a fixed element size (e.g. single byte ANSI, UCS-2 or UTF-32) then the indexing is fast, needing only to multiply the index by the element size to find the offset to the requested code point.

When the element size is variable (e.g. UTF-8 or UTF-16), the string must be scanned to locate the requested code point. However, due to the self-synchronising nature of UTF-8 and UTF-16 encodings, the encoding does not have to be interpreted in order to achieve this.

- With UTF-8, it is only necessary to perform a byte scan looking for the start of each code point and to count the code points found until the i*th* element is located.
- With UTF-16, it is only necessary to scan each successive two byte pair, treating any surrogate pair found as a single code point.

Comparison and Sorting

Comparison of UniChar and UniString variables is permitted. TUniStrings sorting, and character and string comparison is performed according to the default unicode collation sequence. This is initially set to that for the current locale but may be overridden programmatically.

A collation sequence applies to Unicode Characters and is independent of encoding methods. Therefore, the same default collation sequence applies regardless of the encoding method used for a given UniString. The Unicode Collation Algorithm is used as the reference model for UniChar and UniString comparison.

The operators '=', '<>', '<', '<=', '>', '>=' are all performed with respect to the current UniChar collation sequence and apply to:

- the comparison of two UniChar characters or UniStrings.
- The comparison of a UniChar with any other character type
- The comparison of a UniString with any other string type.

Given that the collation sequence is defined for the Unicode Character Set and is invariant between encoding methods, there is no implied transliteration for any of the above comparisons. It is thus possible for valid comparisons to take place between characters or strings which cannot be transliterated into the other's encoding method.

This is not to say that compile or run time optimisations may not take place. For example, when comparing two UniStrings that are encoded as ASCII then a simple byte level comparison may be performed.

Standard RTL functions such as CompareText will also require UniString variants.

The 'POS' Function

The standard “pos” function when applied to a UniString will return index of the Unicode Character at which the substring is found. It is defined as:

```
function Pos(
  const c: UniChar;
  const s: UniString
):SizeInt

function Pos(
  const substr: UniString;
  const s: UniString
):SizeInt
```

The implementation of POS is expected to be almost as efficient for UTF encoded strings as it is for single byte encodings.

- In a single byte encoding, Pos is a bytewise comparison of the string 's' looking for either the first occurrence of 'c' in the string, or the first match for the substr in 's'. Successive bytes are scanned in 's' in order to achieve this.
- When 's' is encoded in a UTF encoding, the scan is for successive code points rather than bytes.
  - When the search is for a single character 'c':
    - if the Unicode Character given by 'c' cannot be encoded using the encoding method used for 's', then Pos returns zero to indicate no match.
    - Otherwise, 'c' is encoded using the encoding method used for 's', and byte scan performed of 's' for the first occurrence of the encoded value of 'c'. This takes advantage of the self-synchronising nature of a UTF stream, avoiding having to interpret the stream whilst scanning it. All UTF encodings guarantee that code point matches can only take place at code point boundaries. If a match is found then Pos returns the index of the first code point match for 'c'. Otherwise, Pos returns zero if no match is found.
  - When the search is for a substring 'substr':
    - if the 'substr' does not use the same encoding method as 's' and 'substr' cannot be transliterated into the encoding method used by 's', then Pos returns zero to indicate no match.
    - If the 'substr' uses the same encoding method as 's', or can be transliterated into the encoding method used by 's', then a byte scan is performed performed of 's' looking

for the first substring match for substr (when encoded in the same encoding method as 's'), if any. If a match is found then Pos returns the index of the first code point match for 'substr'. Otherwise, Pos returns zero if no match is found.

String copy, insert and delete

The standard copy, insert and delete procedures should all apply to UniStrings.

In each case, the “index” refers to the Unicode Character number and not the byte number.

In each case, the “length” refers to the number of Unicode Characters in the string which may differ from the number of bytes.

Input/Output

Input/Output can be performed using Pascal read/readln or write/writeln from “Text” sources and sinks or using TStream objects.

Readln and Writeln

The standard Pascal read/readln and write/writeln operate on “Text” files. Currently, each Text file may have an ANSI code page associated with it. By default this is the DefaultSystemCodePage. The code page can be changed by a call to SetTextCodePage.

- A UniString may be used as an argument to a call to read or readln on a “Text” File. It is handled the same as any other type of string argument. On return, the encoding method for the UniString is set to the code page assigned to the “Text” File and the value of the string is set to the text string read from the file using the file's encoding method.

- A UniString may be used as an argument to a call to write or writeln, and the contents of the UniString written to the “Text” File. If the UniString's encoding method is different from that specified for the “Text” File then it is first transliterated into the encoding method given by the “Text” File's code page. If the transliteration fails then an exception is raised.

TStream

An additional read/write property is added to a TStream. This is:

```
property CodePage: TSystemCodePage;
```

When a stream is created it is set to the DefaultStreamCodePage and can be changed at any time.

The following methods should be added to TStream.

```
Procedure TStream.WriteUniString(const s: UniString);
```

The above follows the TStream.WriteAnsiString model and writes to the stream a four byte length indicator (byte count) followed by the value of string 's' using the encoding method identified by the stream's code page. If necessary, the value of 's' is transliterated to the stream's code page prior to it being written to the stream.

```
Procedure TStream.WriteRawUniString(const s: UniString);
```

The above writes to the stream the value of string 's' using the encoding method identified by the stream's code page with no length indicator. If necessary, the value of 's' is transliterated to the stream's code page prior to it being written to the stream.

```
const DefaultLineSeparator = #U+000A;

Procedure TStream.WriteUniStringLn(const s: UniString;
                            separator: UniString = DefaultLineSeparator);
```

The method writes to the stream the value of string 's' using the encoding method identified by the stream's code page with no length indicator, followed by the separator character(s). By default this is the Unicode Character for a Line Break encoded according to the stream's code page. If necessary, the value of 's' is transliterated to the stream's code page prior to it being written to the stream.

```
Function TStream.ReadUniString: UniString;
```

The above function follows the TStream.ReadAnsiString model and reads a four byte length indicator (byte count) from the stream, followed by the given number of bytes. It returns a UniString consisting of the bytes read set to the encoding method given by the stream's code page and with an encoding type set to the stream's code page. No transliteration takes place as the stream is assumed to be encoded using the stream's code page.

```
Function TStream.ReadRawUniString: UniString;
```

The above function reads the remainder of the stream and returns a UniString containing the bytes read set to the encoding method given by the stream's code page, and with an encoding type set to the stream's code page. No transliteration takes place as the stream is assumed to be encoded using the stream's code page.

```
Function TStream.ReadUniStringLn(separator: UniString = DefaultLineSeparator)
                  UniString;
```

The above function scans the stream as Unicode Characters encoded using the encoding method given by the stream's code page until the given “separator” character(s) is/are found. It returns a UniString comprising all characters read up to but excluding the separator and set to the encoding method given by the stream's code page. For example, this function may be used to read the next line from the stream. No transliteration takes place as the stream is assumed to be encoded using the stream's code page.

TUniStrings

The TUniStrings type is almost functionally identical to TStrings. It differs in identifying the encoding method used for the strings that it contains. The following additional methods and properties are defined. Otherwise, existing TStrings methods and properties with type string are replaced with UniString equivalents and type char with UniChar equivalents.

```
TUniStrings = class(TPersistent)
private
  function StringCodePage: TSystemCodePage;
  procedure SetCodePage(CodePage: TSystemCodePage);
public
  constructor Create(codepage: TSystemCodePage =  DefaultEncodingMethod);
```

```
    property CodePage: TSystemCodePage read StringCodePage write SetCodePage;
  end;
```

The revised constructor allows for the common encoding method to differ from the platform default.

Changing the TUniStrings code page results in transliteration of all strings it contains into the new encoding method.

A TUniStringList will also be required, similar to TStringList.

Note: the TUniStrings.LoadFromStream and TIniStrings.SaveToStream may be used as methods to read and write strings in a TUniStrings from and to a text stream using ReadUniStringLn and WriteUniStringLn, respectively, using the default line separator. In the former case, on return, the TUniStrings code page will be set to the TStream's code page.

External APIs

UniStrings may need to be used with external APIs written in 'C'. For example, under Linux these will typically use null terminated PAnsiChar and, under Windows, PWideChar types. UniStrings may be coerced to either string type and this coercion should be implicit.

With input parameters, it is the programmer's responsibility to ensure that the string is encoded correctly for the API – typically by a call to “SetCodePage” prior to passing the string to the API function. For example, Linux APIs usually expect the string to be UTF-8 encoded. The DefaultEncodingMethod global variable should normally identify the correct codepage for system APIs.

With output parameters, both the codepage and the string length (in unicode characters) must be set correctly by calls to “SetCodePage” and “SetLength” prior to passing the string to the API function.

For example, if an external 'C' function is defined as:

```
  const char* myfunc(const char* inparam, char* outparam,
                                               int max_outparambytes);
```

and, assuming that the API is specified to use UTF-8 character strings and is hence translated into a Pascal external definition as:

```
  function myfunc(const inparam: PAnsiChar; outparam: PAnsiChar;
                             max_outparambytes: integer): PAnsiChar; cdecl; external;
```

then the following Pascal wrapper function could be defined to call it:

```
  function myfunc_wrapper(inparam: UniString; var outparam: UniString;
                                        max_chars: integer = 256): UniString;
  begin
    SetCodePage(inparam,cp_utf8);
    SetCodePage(outparam,cp_utf8);
    SetLength(outparam,max_chars);
    SetString(Result, cp_utf8, myfunc(inparam,outparam,bytelength(outparam)));
  end;
```

In the above, the first call to SetCodePage ensures that the input parameter is encoded in UTF-8 and forces transliteration if it is not. The second call to SetCodePage, ensures that the output parameter is also UTF-8 and sets it to the empty string if it was uninitialised. The call to SetLength reserves space for up to max_chars (default 256)characters plus a null terminator.

The call to “myfunc” then relies on implicit coercion to pass both inparam and outparam as pointers to their internal buffers. Both have UTF-8 as their codepage and hence the coercion to PAnsiChar should not cause an exception. The third function parameter is the maximum length in bytes that can be stored in the outparam. This is obtained by a call to the “bytelength” function (see above). Finally, the function returns a pointer to an null terminated string in some internal buffer. The SetString procedure is used to assign the returned string to the Result variable.

In the above example, a wrapper function is used as a convenience. The calls to SetCodePage, SetLength and SetString could occur in line, but it is clearer if they are located in a wrapper function. In practice, such a wrapper function may well have the same name as the external function (perhaps defined as part of a class) with a unit namespace used to identify the external function.

Command Line Parameters and Environment Variables

The “ParamStr” and “GetEnvironmentVariable” functions should both return a UniString encoded in a platform specific way. Typically on Linux this will be UTF-8 and UTF-16 on Windows.

Given the assignment rules specified above, UniString variations of these functions should be able to replace the existing functions without affecting legacy programs.

Compiler Modes and Delphi Compatibility

The intended long term goal is that UniChar becomes a synonym for “char” and likewise, UniString and TUniStrings for “string” and “TStrings”, respectively. In order not to break existing implementations that, for example, use “char” as a “byte”, a compiler switch should be defined to enable their use as synonyms.

Unless Delphi also implements this proposal, UniChar, etc. will not be available when Delphi Compatibility is a goal and it is hence unlikely that “char = unichar” will ever be enabled by default under “Mode Delphi”.

Implementation Notes

1. The implementation of an AnsiString is very similar to that proposed for a UniString and it could be that the underlying implementation is identical except that the compiler can make assumptions about the encoding method used to make optimisations. The main difference is in indexing where a UniString index returns a code point, while an AnsiString index returns a byte.

2. Similar comments apply to the UTF16String and UnicodeString type where the updated implementation could be identical to that for a UniString except that the compiler is allowed to assume that the string encoding is constrained to an array of single UTF-16 code units. However, there is again an issue over indexing as, with this string type, an indexed UniString returns the indexed UniChar, which is not necessarily the same as the same numbered element in a UTF16String or a UnicodeString if they are used for UTF-16 with surrogate pairs.

3. On the other hand, the WideString type cannot be derived from a UniString as this must be Windows API compatible.

Tony Whyman, MWA, tony.whyman@mccallumwhyman.com
4 October 2017