White Paper

embarcadero

# Delphi and Unicode

By Marco Cantù, Delphi Product Manager

Embarcadero Technologies

Updated December 2013 (Original November 2008)

**Americas Headquarters**
100 California Street, 12th Floor
San Francisco, California 94111

**EMEA Headquarters**
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

**Asia-Pacific Headquarters**
100 Clarence Street
Sydney, NSW 2000
Australia

# Introduction: Delphi and Unicode

This white paper has been written to help developers that are using versions of Delphi or RAD Studio that were released previous to the launch of Delphi 2009. If you are looking for more information on why and how to Unicode-enable a codebase that has been developed using a pre-Unicode enabled version, please read on.

A significant new feature introduced in 2008 with the release of Delphi 2009 was the complete support for the Unicode character set. While Delphi applications written exclusively for the English language and based on a 26-character alphabet were already working fine and will keep working fine in versions that have arrived since Delphi 2009, applications written for most other languages spoken around the world will have a distinct benefit by this change.

This is true for applications written in Western Europe or South America, that used to work fine only within a specific locale, but it is a large benefit for applications written in other parts of the world. Even if you are writing an application in English, consider it is now easier to translate and localize, and that it can now operate on textual data written in any language, including database memo fields with texts in Arabic, Chinese, Japanese, Cyrillic, to name just a few of the world languages supported by Unicode with a simple, uniform, and easy to use character set.

With the Windows operating system providing extensive support for Unicode at the API level, the introduction of Unicode support for Delphi opened up new markets for developers to both sell programs and to develop new specific applications.

As we will see in this white paper, for developers that have not yet adopted Unicode in their applications, there are some new concepts to learn and a few caveats, but the changes open up many opportunities. And in case you need to improve compatibility, you can still keep part of your code to use the traditional string format. But let's not rush through the various topics, and rather start from the beginning. One final word of caution: the concepts behind Unicode and some of the new features introduced here take some time to learn, but you can certainly start using Delphi to convert your existing pre-Delphi 2009 applications right away, without needing the gory details. Unicode enabling is much easier than it might look!

# What is Unicode?

Unicode is the name of an international character set, encompassing the symbols of all written alphabets of the world, of today and of the past, plus a few more. Unicode includes also technical symbols, punctuations, and many other characters used in writing text, even if not part of any alphabet. The Unicode standard (formally referenced as "ISO/IEC 10646") is defined and documented by the Unicode Consortium, and contains over 100,000 characters. Their main web site is located at: http://www.unicode.org.

As the adoption of Unicode was a central element of the Delphi 2009 release there are many points to consider.

The idea behind Unicode (which is what makes it simple) is that every single character has its own unique number (or code point, to use the proper Unicode term). I don't want to delve into the complete theory of Unicode here, but only highlight its key points.

## UNICODE TRANSFORMATION FORMATS

The confusion behind Unicode (what makes it complex) is that there are multiple ways to represent the same code point (or Unicode character numerical value) in terms of actual storage, or of physical bytes. If the only way to represent all Unicode code points in a simple and uniform way was to use four bytes for each code point (in Delphi the Unicode Code Points can be represented using the UCS4Char data type) most developers would perceive this as too expensive in memory and processing terms.

Few people know that the very common "UTF" term is the acronym of Unicode Transformation Format. These are algorithmic *mappings*, part of the Unicode standard, that map each code point (the absolute numeric representation of a character) to a unique sequence of bytes representing the given character. Notice that the mappings can be used in both directions, converting back and forth different representations.

The standard define three of these encodings or formats, depending on how many bits are used to represent the initial part of the set (the initial 128 characters): 8, 16, or 32. It is interesting to notice that all three forms of encodings need at most 4 bytes of data for each code point.

- UTF-8 transforms characters into a variable-length encoding of 1 to 4 bytes. UTF-8 is popular for HTML and similar protocols, because it is quite compact when most characters (like markers in HTML) fall within the ASCII subset.
- UTF-16 is popular in many operating systems (including Windows) and development environments (like Java and .NET). It is quite convenient as most characters fit in two bytes, reasonably compact, and fast to process.
- UTF-32 makes a lot of sense for processing (all code points have the same length), but it is memory consuming and has limited practical usage. Another problem relates with multi-byte representations (UTF-16 and UTF-32) is which of the bytes comes first. According to the standard, all forms are allowed, so you can have a UTF-16 BE (big-endian) or LE (little-endian), and the same for UTF-32.

## Byte Order Mark

Files storing Unicode characters often use an initial header, called Byte Order Mark (BOM) as a signature indicating the Unicode format being used and the byte order form (BE or LE). The following table provides a summary of the various BOM, which can be 2, 3, or 4 bytes long:

| | |
|---|---|
| 00 00 FE FF | UTF-32, big-endian |
| FF FE 00 00 | UTF-32, little-endian |
| FE FF | UTF-16, big-endian |
| FF FE | UTF-16, little-endian |
| EF BB BF | UTF-8 |

## Unicode in WIN32

Since the early days, the Win32 API (which dates back to Windows NT) has included support for Unicode characters. Most Windows API functions have two versions available, an ASCII version marked with the letter A and a wide-string version marked with the letter W. As an example, the following is a small snippet of Windows.pas from Delphi 2009 onwards:

```
function GetWindowText(hWnd: HWND; lpString: PWideChar;
   nMaxCount: Integer): Integer; stdcall;
function GetWindowTextA(hWnd: HWND; lpString: PAnsiChar;
   nMaxCount: Integer): Integer; stdcall;
function GetWindowTextW(hWnd: HWND; lpString: PWideChar;
   nMaxCount: Integer): Integer; stdcall;

function GetWindowText; external user32
   name 'GetWindowTextW';
function GetWindowTextA; external user32
   name 'GetWindowTextA';
function GetWindowTextW; external user32
   name 'GetWindowTextW';
```

The declarations are identical but use either PAnsiChar or PWideChar to refer to strings. Notice that the plain version with no string format indication is just a placeholder for one of them, in past versions of Delphi invariably the 'A' version, while in Delphi 2009 the default became the 'W' version, as you can see above.

# Char is Now WideChar

For some time, Delphi included two separate data types representing characters:

**AnsiChar**, with an 8-bit representation (accounting for 256 different symbols), interpreted depending on your code page;

**WideChar**, with a 16-bit representation (accounting for 64K different symbols).

In this respect, nothing changed with Delphi 2009. What is different is that the **Char** type used to be an alias of **AnsiChar** and is now an alias of **WideChar**. Every time the compiler sees Char in your code, it reads **WideChar**. Notice that there is no way to change this new compiler default. (As with the string type, the Char type is mapped to a specific data type in a fixed and hard- coded way. Developers have asked for a compiler directive to be able to switch, but this would cause a nightmare in terms of QA, support, package compatibility, and much more. You still have a choice, as you can convert your code to use a specific type, such as **AnsiChar**.)

This is quite a change, impacting a lot of source code and with many ramifications. For example, the PChar pointer is now an alias of PwideChar, rather than PAnsiChar, as it used to be.

## CHAR AS AN ORDINAL TYPE

The new *large* Char type is still an ordinal type, so you can use Inc and Dec on it, write for loops with a Char counter, and the like.

```
var
  ch: Char;
begin
  ch := 'a';
  Inc (ch, 100);
  ...
  for ch := #32 to High(Char) do
    str := str + ch;
```

The only thing that might get you into some (limited) trouble is when you are declaring a set based on the entire Char type:

```
var
  CharSet = set of Char;
begin
  charSet := ['a', 'b', 'c'];
  if 'a' in charSet then
    ...
```

In this case the compiler will assume you are porting existing code to Unicode, decide to consider that Char as an AnsiChar (as a set can only have 256 elements at most) and issue a warning message:

```
W1050 WideChar reduced to byte char in set expressions.  Consider
using 'CharInSet' function in 'SysUtils' unit.
```

The code will probably work as expected, but not all existing code will easily map, as it is not possible to obtain a set of all the characters any more. If this is what you need, you'll have to change your algorithm (possibly following what's suggested by the warning).

If what you are looking for, instead, is to suppress the warnings (compiling the five lines of code above causes two of them) you can write:

```
var
  charSet: set of AnsiChar; // suppress warning
begin
  charSet := ['a', 'b', 'c'];
  if AnsiChar('a') in charSet then // suppress warning
...
```

## CONVERTING WITH CHR

Notice also that you can convert a numeric value to a character using a type cast to AnsiChar or WideChar, but also relying on the classic Pascal technique, the use of the Chr *compiler magic* function (which can be considered as the opposite of Ord). This standard magic function has been expanded to take a word as parameter, rather than a byte.

Although, unlike character literals, calls to Chr are now always interpreted in the Unicode realm. So if you port code like:

```
Chr (128)
```

from Delphi 2007 you might be in for a surprise. If you use #128, instead, you may get a different result, depending on your code page.

## 32-BIT CHARACTERS

Although the default Char type is now mapped to WideChar, it is worth noticing that Delphi defines also a 4-byes character type, UCS4Char, defined in the System unit as:

```
type
```

```
UCS4Char = type LongWord;
```

While this type definition and the corresponding one for UCS4String (defined as an array of UCS4Char) were already in Delphi 2007, the relevance of the UCS4Char data type in Delphi 2009 onwards comes from the fact it is now significantly used in several RTL routines, including those of the new Character unit discussed next.

## THE NEW CHARACTER UNIT

To better support the new Unicode characters (and also Unicode strings, of course) Delphi 2009 introduced a brand new RTL unit, called Character. The unit defines the TCharacter sealed class, which is a basically collection of static class functions, plus a number of global routines mapped to the public (and some of the private) functions of the class.

The unit also defines two interesting enumerated types. The first is called TUnicodeCategory and maps the various characters in broad categories like control, space, uppercase or lowercase letter, decimal number, punctuation, math symbol, and many more. The second enumeration is called TUnicodeBreak and defines the family of the various spaces, hyphen, and breaks.

The TCharacter sealed class has over 40 methods that either work on a stand-alone character or one within a string for:

- Getting the numeric representation of the character (GetNumericValue).
- Asking for the category (GetUnicodeCategory) or checking it against one of the various categories (IsLetterOrDigit, IsLetter, IsDigit, IsNumber, IsControl, IsWhiteSpace, IsPunctuation, IsSymbol, and IsSeparator)
- Checking if it is lowercase or uppercase (IsLower and IsUpper) or converting it (ToLower and ToUpper)
- Verifying if it is part of a UTF-16 surrogate pair (IsSurrogatePair, IsSurrogate, IsLowSurrogate, and IsHighSurrogate)
- Converting it to and from UTF32 (ConvertFromUtf32 and ConvertToUtf32)

The global functions are almost an exact match of these static class methods, some of which correspond to existing Delphi RTL functions even if generally with different names. There are overloads of some of the basic RTL functions working on characters, with extended versions that call the proper Unicode-enabled code. For example, you can write the following code for trying to convert an accented letter to uppercase:

```
var
  ch1: Char;
  ch2: AnsiChar;
begin
  ch1 := 'ù';
  Memo1.Lines.Add ('WideChar');
  Memo1.Lines.Add ('UpCase ù: ' + UpCase(ch1));
  Memo1.Lines.Add ('ToUpper ù: ' + ToUpper (ch1));
  ch2 := 'ù';
  Memo1.Lines.Add ('AnsiChar');
  Memo1.Lines.Add ('UpCase ù: ' + UpCase(ch2));
  Memo1.Lines.Add ('ToUpper ù: ' + ToUpper (ch2));
```

The traditional Delphi code (the UpCase on the AnsiChar version) handles ASCII characters only, so it won't convert the character (The same is true for the UpperCase function, which handles only ASCII, while AnsiUpperCase handles everything in Unicode, despite the name.). The behavior doesn't change (probably for backward compatibility reasons) if you pass a WideChar to it. The ToUpper function works properly (it ends up calling the CharUpper function of the Windows API). This is the output of running the code above:

```
WideChar
UpCase ù: ù
ToUpper ù: Ù
AnsiChar
UpCase ù: ù
ToUpper ù: Ù
```

Notice you can keep your existing Delphi code, with the UpCase call on a Char, and it will keep the standard Delphi behavior.

For a better demo of the specific Unicode-related features introduced by the Characters unit, you can see the following code, which defines a string including Unicode code point $1D11E, that is *musical symbol G clef*:

```
var
  str1: string;
begin
  str1 := '1.' + #9 + ConvertFromUtf32 (128) +
    ConvertFromUtf32($1D11E);
```

The program then makes the following tests (all returning True) on the various characters of the string:

```
  TCharacter.IsNumber(str1, 1)
  TCharacter.IsPunctuation (str1, 2)
  TCharacter.IsWhiteSpace (str1, 3)
```

```
TCharacter.IsControl(str1, 4)
TCharacter.IsSurrogate(str1, 5)
```

Finally notice that the IsLeadChar function of SysUtils has been modified to handle Unicode surrogate pairs, as well as other related functions used to move to the next character of a string and the like.

# OF STRING AND UNICODESTRING

The change in the definition of the Char type is important because it is tied to the change in the definition of the string type. Unlike characters, though, string is mapped to a brand new data type that didn't exist before, called UnicodeString. As we'll see, its internal representation is also quite different from that of the *classic AnsiString* type (I'm using the specific terms *classic AnsiString* type, to refer to the string type as it used to work from Delphi 2 until Delphi 2007; the AnsiString type is still part of Delphi 2009 onwards, but it has a modified behavior, so when referring its past structure I'll use the term *classic AnsiString).*

As there was already a WideString type in the language, representing strings based on the WideChar type, why bother defining a new data type? WideString was (and still is) not reference counted and is extremely poor in terms of performance and flexibility (for example, it uses the Windows global memory allocator rather than the native FastMM4).

Like AnsiString, UnicodeString is reference counted, uses copy-on-write semantics and is quite performant. Unlike AnsiString, UnicodeString uses two-bytes per character and is based on UTF-16. Actually UTF-16 is a variable length encoding, and at times UnicodeString used two WideChar surrogate elements (that is, four bytes) to represent a single Unicode code point.

The string type is now mapped to UnicodeString in a hard-coded way as is the Char type and for the same reasons. There is no compiler directive or other trick to change that. If you have code that needs to continue to use the string type, just replace it with an explicit declaration of the AnsiString type.

## THE INTERNAL STRUCTURE OF STRINGS

One of the key changes related to the new UnicodeString type is its internal representation. This new representation, however, is shared by all reference-counted string types, UnicodeString and AnsiString, but not by the non-reference counted string types, ShortString and WideString.

The representation of the *classic AnsiString* type was the following:

| -8 | -4 | String reference address |
|---|---|---|
| Ref count | length | First char of string |

The first element (counting backwards from the beginning of the string itself) is the Pascal string length, the second element is the reference count. In Delphi 2009 the representation for reference-counted strings becomes:

| -12 | -10 | -8 | -4 | String reference address |
|---|---|---|---|---|
| Code page | Elem size | Ref count | length | First char of string |

Beside the length and reference count, the new fields represent the element size and the code page. While the element size is used to discriminate between AnsiString and UnicodeString, the code page makes sense in particular for the AnsiString type (as it has worked since Delphi 2009), as the UnicodeString type has the fixed code page 1200.

A corresponding support data structure is declared in the implementation section of System unit as:

```
type
  PStrRec = ^StrRec;
  StrRec = packed record
    codePage: Word;
    elemSize: Word;
    refCnt: Longint;
    length: Longint;
end;
```

As it is in the implementation section you cannot use it in your code, which is understandable for an internal data structure that's implementation specific and subject to change. There are helper functions to access the information you'll generally need to use.

With the overhead of a string going from 8 bytes to 12 bytes, one might wonder if a more compact representation wouldn't be more effective, although the newer fields are more compact than the traditional ones (that could be changed only at the expense of compatibility). This is a classic trade-off between memory and speed: by storing data in different memory locations (and not using portions of a single location) you gain extra runtime speed, although this is costing extra memory for each and every string you create.

While in the past you had to use low-level pointer-based code to access to the reference count, the Delphi 2009 RTL added some handy functions to access the various string metadata:

```
function StringElementSize(const S: UnicodeString): Word;
function StringCodePage(const S: UnicodeString): Word;
function StringRefCount(const S: UnicodeString): Longint;
```

There is also a new helper functions in the SysUtils unit, called ByteLength, that returns the size of a UnicodeString in bytes ignoring the StringElementSize attributes (so, oddly enough, it won't work with string types other than UnicodeString).

As an example, you can create a string and ask for some information about it:

```
var
  str1: string;
begin
  str1 := 'foo';
  Memo1.Lines.Add ('SizeOf: ' + IntToStr (SizeOf (str1)));
  Memo1.Lines.Add ('Length: ' + IntToStr (Length (str1)));
  Memo1.Lines.Add ('StringElementSize: ' +
    IntToStr (StringElementSize (str1)));
  Memo1.Lines.Add ('StringRefCount: ' +
    IntToStr (StringRefCount (str1)));
  Memo1.Lines.Add ('StringCodePage: ' +
    IntToStr (StringCodePage (str1)));
if StringCodePage (str1) = DefaultUnicodeCodePage then
    Memo1.Lines.Add ('Is Unicode');
  Memo1.Lines.Add ('Size in bytes: ' +
    IntToStr (Length (str1) * StringElementSize (str1)));
  Memo1.Lines.Add ('ByteLength: ' +
    IntToStr (ByteLength (str1)));
```

This program produced output similar to the following:

```
SizeOf: 4
Length: 3
StringElementSize: 2
StringRefCount: -1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

The code page returned by a UnicodeString is 1200, a number stored in the global variable DefaultUnicodeCodePage. In the code above (and its output) you can clearly notice that there isn't a direct call to determine the length of a string in bytes, since Length returns the number of characters.

Of course, you can (in general) multiply this by the size in bytes of each character, using the expression:

```
Length (str1) * StringElementSize (str1)
```

Not only can you ask a string for information, but you can also change some of it. A low-level way to convert a string is to call the SetCodePage procedure (an operation applicable only to a RawByteString type, as we'll see), which can either only adjust the code page to the real one or perform a full string conversion. I'll use this procedure in the section "String Conversions".

## UNICODESTRING AND UNICODE

Needless to say the new string type (or new UnicodeString type, to be more precise) maps to the Unicode character set. However, the question becomes, "which flavor of Unicode?"

It should not be surprising to learn that the new string type uses UTF-16. More precisely, the UnicodeString type in stored in memory as a UTF-16 string with a little endian representation, or UTF-16 LE. This makes a lot of sense for many reasons, the most significant being that this is the native string type managed by the Windows API in recent versions of the operating system.

As we've seen in the section covering the WideChar type in Delphi 2009, the new TCharacter support class (not used for WideChar but also for UnicodeString processing) has full support for UTF-16 and surrogate pairs. What I didn't mention in the section is that this has the noticeable side effect of making the number of WideChar elements of a string different from the number of Unicode code points it contains, as a single Unicode code point can be represented by a surrogate pair (that is, two WideChar).

A way to create a string with surrogate pairs is to use the ConvertFromUtf32 function that returns a string with the surrogate pair (two WideChar) in the proper circumstances, like the following:

```
var
  str1: string;
begin
  str1 := 'Surr. ' + ConvertFromUtf32($1D11E);
```

Now if you ask for the string length, you'll get 8, which is the number of WideChar, but not the number of logical Unicode code points in the string. If you print the string you get the proper effect (well, at least Windows will generally show one square block as placeholder of the surrogate pair, rather than two).

By the way, in the code of ConvertFromUtf32 (or more precisely in the ConvertFromUtf32 class method of the TCharacter class it calls) you can see the actual algorithm used for mapping Unicode code points into surrogate pairs. Interesting reading if you are interested in the details.

A related issue is what happens when looping on each character of the string. A standard for loop or a **for-in** cycle will just let you work on each WideChar element of the string, not each logical Unicode code point. So you might have to use a while loop based on the **NextCharIndex** function or adapt the for loop checking for surrogates:

```
if TCharacter.IsHighSurrogate (str1 [I]) then
  Memo1.Lines.Add (str1 [I] + str1 [I+1])
```

**However, in most cases you can assume to work with the BMP (Basic Multilingual Plane) that treats each WideChar of a Unicode string as a single code point.**

## THE UCS4STRING TYPE

There is also another string type that you can use to handle a series of Unicode code points, the UCS4String type. This data type represents a dynamic array of 4-bytes characters (the UCS4Char type). As such, it has no reference counting or copy-on-write support, and very little RTL support.

Although this data type (that was already available in Delphi 2007) can be used in specific situations, it is not particularly suited for general circumstances. It certainly can be a memory waster, as not only strings use 4 bytes per character, but you can end up with multiple copies in memory.

# THE MANY STRING TYPES

Along with the introduction of the new UnicodeString type, the updated internal representation shared by all string types (including the AnsiString type) makes room for some extra improvements in string management. The Delphi R&D team took advantage of this new internal representation (and all the work they did at the compiler level to enhance string management) to actually provide you with multiple data types and even a brand new string type definition mechanism.

The predefined string types, in addition to UnicodeString, are:

- **AnsiString** is a single-byte-per-character string type based on the current code page of the operating system, closely matching the *classic AnsiString* of past versions of Delphi;

- UTF8String is a string based on the variable character length UTF8 format;
- RawByteString is an array of characters with no code page set, on which no character conversion is accomplished by the system (thus partially resembling the *classic AnsiString*, when used as a pure character array).

The type definition mechanism is revealed as you look at the definition of these new string types:

```
type
  UTF8String = type AnsiString(65001);
  RawByteString = type AnsiString($FFFF);
```

In this next section I'll cover the AnsiString and custom string types and then the UTF8String type. I'll focus on RawByteString in the following section covering string conversions, as you generally use this string type to avoid conversions.

## THE NEW ANSISTRING TYPE

Differently from the past, the new AnsiType string carries one further piece of information, the code page of the characters in the string. The DefaultSystemCodePage variable defaults to CP_ACP, the current Windows code page, but it could be modified by calling the special procedure, SetMultiByteConversionCodePage. You can do this to force an entire program to work (by default) with characters in a given code page (that the operating system installation must support, of course).

In general, instead, you'd either stick to the current code page or change it for individual strings, calling the SetCodePage procedure (introduced earlier while talking about characters and code pages). This procedure can be called in two different ways. In the first case, you change the code page of a string (maybe loaded by a separate file or socket) because you know its format. In the second case, you can call it to convert a given string (something that happens automatically when assigning a string to one of a different code page, as discussed later).

Although you can keep using the AnsiString type to have a more compact in-memory representation of strings, in most cases you'd really want to convert your code to using the new UnicodeString type, that is, keep your strings declared with the generic string type. Still, there are circumstances in which using a specific string type is necessary. For example, cases such as loading or saving files, moving data from and to a database, using Internet protocols where the code must remain in an 8-bit per character format. In all those cases convert your code to use AnsiString.

## CREATING A CUSTOM STRING TYPE

Besides using the new AnsiString type, which is tied to the default code page used when compiling the application, you can use the same mechanism to define your own custom string type. For example, you can define a Latin-1 string type by writing:

```
type
  Latin1String = type AnsiString(28591);
procedure TFormLatinTest.btnNewTypeClick(
  Sender: TObject);
var
  str1: Latin1String;
begin
  str1 := 'a string with an accent: Cantù';
  Log ('String: ' + str1);
```

You can use this string type as any other one, but it will be tied to a specific code page. So if you use this string type, when you convert a Latin1String to a UnicodeString (for example, to display it in a call to Log above), the Delphi compiler will add a conversion call. The last line of the code snippet above has a *hidden* call to _UStrFromLStr, which end up calling more internal functions of the system unit, up to the real conversion operation performed by the MultiByteToWideChar Windows API. This is the sequence of calls:

```
procedure _UStrFromLStr(var Dest: UnicodeString;
  const Source: AnsiString);
procedure InternalUStrFromPCharLen(
  var Dest: UnicodeString; Source: PAnsiChar;
  Length: Integer; CodePage: Integer);
function WCharFromChar(WCharDest: PWideChar;
  DestChars: Integer; const CharSource: PAnsiChar;
  SrcBytes: Integer; CodePage: Integer): Integer;
function MultiByteToWideChar(CodePage, Flags: Integer;
  MBStr: PAnsiChar; MBCount: Integer;
  WCStr: PWideChar; WCCount: Integer): Integer; stdcall;
  external kernel name 'MultiByteToWideChar';
```

The Windows API can perform the proper conversions, but these are potentially lossy conversions, as even some characters available in the various Windows code pages cannot be represented in Latin1. An example would be the Euro currency symbol, another the smart quotes.

The btnNewTypeClick method above continues showing some more details of the string:

```
Log ('Last char: ' + IntToStr (
  Ord (str1[Length(str1)])));
Log ('ElemSize: ' + IntToStr (StringElementSize (str1)));
Log ('Length: ' + IntToStr (Length (str1)));
```

```
Log ('CodePage: ' + IntToStr (StringCodePage (str1)));
```

Running this code produces the following output:

```
Last char: 249
ElemSize: 1
Length: 30
CodePage: 28591
```

To prove that this new custom string type is treated differently than the standard AnsiString type (at least on my computer and with my locale), I've written a test method that adds the same upper end characters (from #128 to #255) to both an AnsiString and a Latin1String, showing them on a Memo in groups:

```
procedure TFormLatinTest.btnCompareCharSetClick(
  Sender: TObject);
var
  str1: Latin1String;
  str2: AnsiString;
  I: Integer;
begin
  for I := 128 to 255 do
  begin
    str1 := str1 + AnsiChar (I);
    str2 := str2 + AnsiChar (I);
end;
for I := 0 to 15 do
  begin
    Log (IntToStr (128 + I*8) + ' - ' +
      IntToStr (128 + I*8 + 7));
    Log ('Lati: ' + Copy (str1, 1 + i*8, 8));
    Log ('Ansi: ' + Copy (str2, 1 + i*8, 8));
  end;
end;
```

The initial part of the output highlights the differences among the two sets (again, the result you'll see might vary depending on your own locale):

```
128 - 135
Lati: ?,f".??
Ansi: €‚ƒ„...†‡
136 - 143
Lati: ^?S<OZ
Ansi: ˆ‰Š‹ŒŽ
144 - 151
Lati: ''"".--
Ansi: ‘’“”•–—
152 - 159
```

```
Lati: ~Ts>ozY
Ansi:  ˜TMš ›œžŸ
```

Having said this, at least at my latitude, a far more interesting example would be to use the code page of a non Latin alphabet, like Cyrillic. As an example, I defined a second custom string type:

```
type
  CyrillicString = type Ansistring(1251);
```

You can use this string in a very similar fashion of the previous code snippet, but the interesting part is to use the high-order characters, those with a numeric value over 127. I've picked a few with a for loop:

```
procedure TFormLatinTest.btnCyrillicClick(
  Sender: TObject);
var
  str1: CyrillicString;
  I: Integer;
begin
  str1 := 'a string with an accent: Cantù';
  Log ('String: ' + str1);
  Log ('Last char: ' + IntToStr (
    Ord (str1[Length(str1)])));
  Log('ElemSize: ' + IntToStr (StringElementSize (str1)));
  Log('Length: ' + IntToStr (Length (str1)));
  Log ('CodePage: ' + IntToStr (StringCodePage (str1)));
  str1 := '';
  for I := 150 to 250 do
    str1 := str1 + CyrillicString(AnsiChar (I));
  Log ('High end chars: ' + str1);
end;
```

The output of this method looks like this:

```
String: a string with an accent: Cantu
Last char: 117
ElemSize: 1
Length: 30
CodePage: 1251
High end chars: ––™љ ›њќћџ  У˘у˘J¤Ґ¦§Е¨©Є«¬–
®І¨°±Іґµ¶ ·е¨№є»jSsi¨АБВГДЕЖЗИИ˘КЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзии˘клмно
прстуфхцчшщъ
```

You can notice that the accented letter has been *converted* to the corresponding non-accented version, as the original value was not available. (The **WideCharToMultiByte** behind the conversion tries to fail gracefully in certain situations. For example, smart quotes degrade into straight quotes instead of question marks and the accented letter of the sample code lost its accent.) The string constant is a Unicode string and the assignment to str1 performs an implicit conversion. In fact, the numeric value of the last character is different.

Also this time the high-end characters are completely different. To obtain the desired effect, consider you have to write the double cast:

```
CyrillicString(AnsiChar (I))
```

If you simply concatenate the characters in the string and convert it afterwards, they'll be treated as Unicode characters.

## MANAGING UTF-8 STRINGS

One of the side effects of the new internal structure for string types, is that we can now also manage strings in the UTF-8 format in a more native way. Unlike the past, when UTF8String was simply an alias of the string type, the new type is now fully recognized: conversions are automatic and all of the existing UTF-8 string manipulation routines have been ported to use the new specific types.

Consider this trivial code:

```
var
  str8: Utf8String;
  str16: string;
begin
  str8 := 'Cantù';
  Memo1.Lines.Add ('UTF-8');
  Memo1.Lines.Add('Length: ' + IntToStr (Length (str8)));
  Memo1.Lines.Add('5: ' + IntToStr (Ord (str8[5])));
  Memo1.Lines.Add('6: ' + IntToStr (Ord (str8[6])));

  str16 := str8;
  Memo1.Lines.Add ('UTF-16');
  Memo1.Lines.Add('Length: ' + IntToStr (Length (str16)));
  Memo1.Lines.Add('5: ' + IntToStr (Ord (str16[5])));
```

As you might expect, the str8 string has a length of 6 (meaning 6 bytes), while the str16 string has a length of 5 (meaning 10 bytes, though). Notice that Length invariably returns the number of string elements, which in case of variable-length representations don't match the number of Unicode code points represented by the string. This is the output of the program:

```
UTF-8
Length: 6
5: 195
6: 185

UTF-16
Length: 5
5: 249
```

The reason is that UTF-8 strings use a variable length implementation, so that characters outside the initial 7-bit ANSI space take at least two characters. This is also the case of the *accented u* above. Assigning the same UTF-8 string to an AnsiString variable, and running similar code, gives the following:

```
ANSI
Length: 5
5: 249
```

However, this time the string length of 5 really means 5 bytes and not only 5 characters.

The support for the UTF-8 format might not as complete as that for UTF-16 , the native string implementation for Delphi 2009, but has been enhanced in a very significant way. There are specific routines for UTF-8 manipulation in the WideStrUtils unit, but also full support for streaming text files in this format (I'll cover the TEncoding class and text file conversions later, in the section "Streams and Encodings"). What's core, though, is the fact you can work on such a string and show it in any control without having to perform an explicit conversion (and having to remember if and when to perform one); that certainly helps a lot.

Even if some operations on UTF-8 strings might be slow, because of extra conversions to and from the UnicodeString type, having a specific data type rather than an alias type not enforced by the compiler makes a lot of difference to any Delphi developer who has to deal with this encoding.

You are also free to write overloaded versions of existing routines (or new ones) using this specific string type to avoid any extra conversion.

# Converting Strings

We've seen you can assign UnicodeString value to an AnsiString or an UTF8String and the proper conversions will take place. Similarly, when you assign an AnsiString with a given code page to another one based on a different code page a conversion happens. You can also

convert a string by assigning to it a different code page, asking for a conversion to take place:

```
type
  Latin1String = type AnsiString(28591);

procedure TFormStringConvert.btnLatin1Click(
  Sender: TObject);
var
  str1: AnsiString;
  str2: Latin1String;
  rbs: RawByteString;
begin
  str1 := 'any string with a €';
  str2 := str1;
  Memo1.Lines.Add (str1);
  Memo1.Lines.Add (IntToStr (Ord (str1[19])));
  Memo1.Lines.Add (str2);
  Memo1.Lines.Add (IntToStr (Ord (str2[19])));
  rbs := str1;
  SetCodePage(rbs, 28591, True);
  Memo1.Lines.Add (rbs);
  Memo1.Lines.Add (IntToStr (Ord (rbs[19])));
end;
```

In both cases above, the conversion is a lossy conversion, because the Euro symbol cannot be represented in the Latin1 code page. Notice the use of the SetCodePage routine, that can be applied only to a RawByteString parameter, hence the assignment. The output you'll get is:

```
any string with a €
128
any string with a ?
63
any string with a ?
63
```

## Conversions May Slow Down The Code

The automatic conversions happening behind the scenes are extremely handy, as the system does a lot of work for you, but if you don't carefully consider what you are doing you might end up with some slow code, because of continuous conversions and string copy operations. Consider the following code:

```
  str1 := 'Marco ';
  str2 := 'Cantù ';
  for I := 1 to 10000 do
```

```
  str1 := str1 + str2;
```

Depending on the actual string type of the two strings, the algorithm can be extremely fast or excruciatingly slow. The demo uses string (that is UnicodeString) in a first run and a combination of AnsiString and UTF8String (the worse possible case, as they'll have to be converted back and forth to the UnicodeString type for each assignment) in a second. This is the result of 10,000 iterations:

```
plain: 00.001
mixed: 01.717
```

Yes, you are reading the right numbers, that's about 1,000 times or three orders of magnitude! If this wasn't bad enough, consider what happens with 50,000 concatenations:

```
plain: 00:00.003
mixed: 00:42.879
```

That's another order of magnitude! (The increase is exponential due to the fact than larger and larger strings need to be re-allocated in memory many times. What slows down the code is only partially the conversion, but mostly the need to create new large temporary strings rather than keep increasing the size of the current one.) In other words, an occasional implicit conversion is fine, but never ever let them happen within a loop or recursive routine!

What is important to know, is that you can compile your program with string conversion warnings enabled (which is actually the default), and see where the compiler adds conversion code. On that single line of code used for concatenating strings of different types you'll get the following warnings:

```
W1057 Implicit string cast from 'UTF8String' to 'string'
W1057 Implicit string cast from 'AnsiString' to 'string'
W1058 Implicit string cast with potential data loss from 'string' to
'UTF8String'
```

The “potential data loss” problem arises because not all strings can be expressed in all formats. For example, if you assign a UnicodeString to an AnsiString there are chances that the operation won't be possible. As string conversion operations are quite common, the corresponding two warnings (Implicit string cast and Implicit string cast with potential data loss) are turned off by default.

With these warnings on you'll see many potential pitfalls, but an average program can have many and even an explicit typecast won't remove them but simply change them to a different set of warnings (Explicit string cast and Explicit string cast with potential data loss). Turn these warnings off when you are done checking!

A fifth similar warning is issued when assigning a string constant to a string, in case some of the characters cannot be converted. The warning in this case is slightly different:

```
[DCC Warning] StringConvertForm.pas(63): W2455 Narrowing given wide
string constant lost information
```

This is a warning you should get rid of, as the operation won't make a lot of sense.

As another example of an implicit (and somewhat hidden) conversion slowing down the program execution, consider the following code snippet:

```
  str1 := 'Marco Cantù';
  for I := 1 to MaxLoop2 do
    str1 := AnsiUpperCase (str1);
```

In case the str1 variable is a UnicodeString all is fine, but in case it is an AnsiString, it will cause two conversions. This is not as bad as in the previous case (because there string is short and a copy of the string is required anyway) but shows a little overhead (for one million iterations):

```
AnsiUpperCase (string): 00:00.289
AnsiUpperCase (AnsiString): 00:00.540
```

## USING RAWBYTESTRING

What if you need pass an AnsiString as parameter to a routine? When the parameter is assigned to a specific string type with an encoding, it will be converted to the proper type, with a potential data loss. That's why Delphi 2009 introduces yet another custom string type, called RawByteString and defined as:

```
type
  RawByteString = type AnsiString($ffff);
```

This is definition creates a string type with no encoding or, to be more precise, with the placeholder $ffff indicating "*no encoding*". A RawByteString can be considered as a string of bytes, which ignores the attached encoding in case of an automatic conversion when assigning to an AnsiString. In other words, when passing a 1-byte per character string as a RawByteString parameter, no conversion is performed, unlike any other AnsiString derived type. You can do a specific conversion by calling the SetCodePage routine, as demonstrated earlier in the section "Converting Strings".

As such, it can become a handy replacement of the string (or AnsiString) type in code that uses strings for generic and custom data processing which you want to keep with a 1-byte per character representation. (Don't be confused by this extended support for 1-byte per character Ansi-compatible strings: the preferred solution is by far to migrate your string processing code to the UnicodeString type. Don't be too tempted by these new extra string types.)

Declaring variables of type RawByteString for storing an actual string should rarely be done. Given the undefined code page, this can lead to undefined behavior and potential data loss. On the other hand if your goal is saving binary data using a string-like memory allocation and representation, you can use the RawByteString in the same way you used AnsiString in past versions of Delphi. Replacing non-string code that used AnsiString with RawByteString is an interesting migration path.

For now, let's focus on a typical example in which you can use the RawByteString type as parameter. If you want to display some information about an 8-bit string, you could write either of the following two declarations (these are methods of the main form of the RawTest demo):

```
  procedure DisplayStringData (str: AnsiString);
  procedure DisplayRawData (str: RawByteString);
```

The code of the two methods is identical (here I've listed only one of the two):

```
procedure TFormRawTest.DisplayRawData(
  str: RawByteString);
begin
  Log ('DisplayRawData(str: RawByteString)');
  Log ('String: ' + UnicodeString(str));
  Log ('CodePage: ' + IntToStr (StringCodePage (str)));
  Log ('Address: ' + IntToStr (Integer (Pointer (str))));
end;
```

Notice that cast to UnicodeString used to display the proper string, which is necessary to avoid the data being treated like a plain AnsiString because of the concatenation of a string literal with a string whose code page is not defined at compile time. (Using Log (str) directly would work, as there is no concatenation involved.)

# STREAMS AND ENCODINGS

If moving all your strings to Unicode within your application, when working with the RTL and VCL, and while invoking the Windows API isn't that hard, things can become a little more complicated as you read and write your strings to and from files. What happens with the TStrings file operations, for example?

Delphi 2009 introduced another brand new class to handle file encodings, called TEncoding and somewhat mimicking the System.Text.Encoding class of the .NET framework. The TEncoding class, defined in the SysUtils unit, has several subclasses representing the encodings automatically supported by Delphi (these are *standard encodings* to which you can add your own):

```
type
  TEncoding = class
    TMBCSEncoding = class(TEncoding)
 TUTF7Encoding = class(TMBCSEncoding)
   TUTF8Encoding = class(TUTF7Encoding)
  TUnicodeEncoding = class(TEncoding)
    TBigEndianUnicodeEncoding = class(TUnicodeEncoding)
```

One object of each of these classes is available within the TEncoding class, as class data, and has a corresponding getter function and class property:

```
type
  TEncoding = class
    ...
  public
    class property ASCII: TEncoding read GetASCII;
    class property BigEndianUnicode: TEncoding
      read GetBigEndianUnicode;
    class property Default: TEncoding read GetDefault;
    class property Unicode: TEncoding read GetUnicode;
    class property UTF7: TEncoding read GetUTF7;
    class property UTF8: TEncoding read GetUTF8;
```

The TEncoding class has methods for reading and writing characters to byte streams, to perform conversions, plus a special function to handle the BOM called GetPreamble. So you can write (anywhere in the code):

```
TEncoding.UTF8.GetPreamble
```

## STREAMING TSTRINGS

The ReadFromFile and WriteToFile methods of the TStrings class can be called with an encoding. If you write a string list to text file without providing a specific encoding, the class will use TEncoding.Default, which uses the internal DefaultEncoding in turn extracted at the first occurrence by the current Windows code page. In other words, if you save a file you'll get the same ANSI file as before.

Of course, you can also easily force the file to a different format, for example the UTF-16 format:

```
Memo1.Lines.SaveToFile('test.txt',
  TEncoding.Unicode);
```

This saves the file with a Unicode BOM or preamble. As you do the corresponding LoadFromFile operation, if you don't specify an encoding, the loading method will end up calling the GetBufferEncoding method of the TEncoding class that will determine the encoding depending on the presence of a BOM (of its absence, in which case it will use the default ANSI encoding).

What if you specify an encoding in LoadFromFile? The encoding you provide will be used for reading the file, regardless of the actual BOM in the file, often producing an error. I'd rather expect an exception in case of such a discrepancy, saving a file with one code page and forcing to upload it with a different one is certainly a developer error. Not having an exception can help in case the encoded file was saved without a BOM, and still should not be considered as an ASCII file, but a UTF one.

But let us focus on the file saving operation. If you don't change the existing Delphi code, your programs will save files as ANSI. If your existing programs don't handle Unicode data, your program and its files will be fully backwards compatible. But what if a program does handle Unicode data? Let's suppose we have a string list with lines written in different languages, like in the following design-time form:

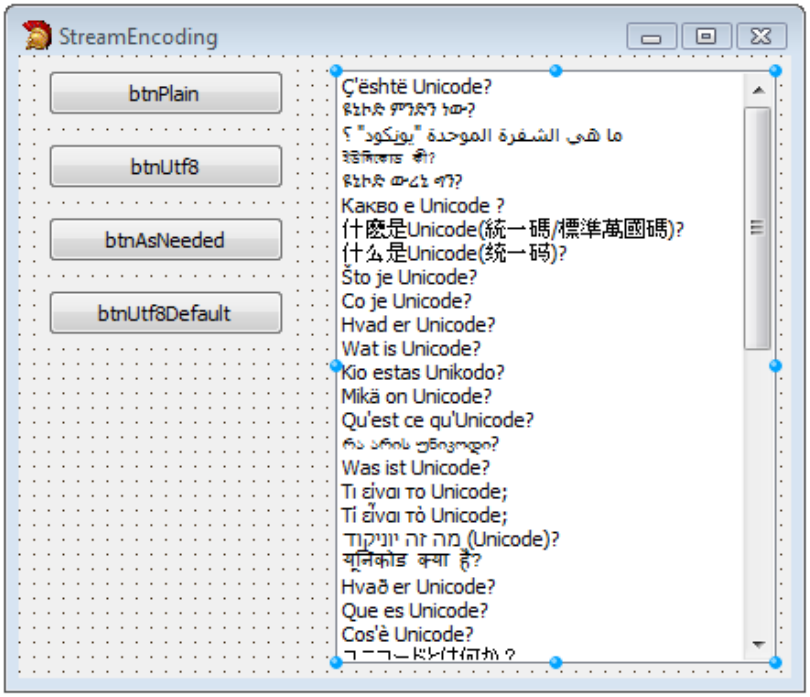


Figure 1 Design Form That Displays Strings in Different Languages

If we have existing Delphi code that saves the string list to a file and reloads it, it would probably look like:

```
procedure TFormStreamEncoding.btnPlainClick(
  Sender: TObject);
var
  strFileName: string;
begin
  strFileName := 'PlainText.txt';
  ListBox1.Items.SaveToFile(strFileName);
  ListBox1.Clear;
  ListBox1.Items.LoadFromFile(strFileName);
end;
```

Needless to say that the effect would be a total disaster, as only a fraction of the characters used have an ANSI representation, so you'll end with lots of question marks in the list box. A simple alternative would be to change the code as in the event handle of the second button of the project:

```
  strFileName := 'Utf8Text.txt';
  ListBox1.Items.SaveToFile(strFileName, TEncoding.UTF8);
```

Again, we don't have to specify an encoding when loading the string list, as Delphi will pick it up from the BOM. If you prefer to save the data as ANSI unless necessary, you could check for the string list content to determine whether to save as ASCII or UTF-8:

```
procedure TFormStreamEncoding.btnAsNeededClick(
  Sender: TObject);
var
  strFileName: string;
  encoding1: TEncoding;
begin
  strFileName := 'AsNeededText.txt';
  encoding1 := TEncoding.Default;
  if ListBox1.Items.Text <>
      UnicodeString (AnsiString(ListBox1.Items.Text)) then
    encoding1 := TEncoding.UTF8;
  ListBox1.Items.SaveToFile(strFileName, Encoding1);
```

This code checks whether you can convert a string to an AnsiString and back to a UnicodeString without losing any content. For a very long string, this double conversion plus comparison would be quite expensive, so you could rather use the following alternative code (which is not as precise, as it relies on a specific code page, but comes close):

```
var
```

```
  ch: Char;
begin
  ...
  for ch in ListBox1.Items.Text do
    if Ord (ch) >= 256 then
    begin
      encoding1 := TEncoding.UTF8;
      break;
    end;
```

Using similar code you could decide which format to use, depending on the situation. It might be a better idea, though, to move all of your files to Unicode encoding (UTF-8 or UTF-16), regardless of the actual data. Using UTF-16 will make the files bigger, but will also reduce the conversions when saving and loading.

However, since there is no way to specify a default conversion, going for Unicode encoding of your files would mean the need to change each and every file save operation... unless we use a trick, changing the standard behavior of the class. Such a *hack* could come in the form of a class helper. Consider the following code:

```
type
  TStringsHelper = class helper for TStrings
    procedure SaveToFile (const strFileName: string);
  end;

procedure TStringsHelper.SaveToFile(
  const strFileName: string);
begin
  inherited SaveToFile (strFileName, TEncoding.UTF8);
end;
```

Notice that inherited here doesn't mean to call a base class but the class helped by the class helper. Now you simply write (or keep your code as):

```
  ListBox1.Items.SaveToFile(strFileName);
```

to save it as UTF8 (or any other encoding of your choice).

# Conclusion: Unicode and The VCL

Having Unicode string support in the Delphi language is thrilling, and having the Win32 APIs remapped to the Wide version opens up a lot of easy migration, but the fundamental change is that the entire RTL and the Visual Component Library (VCL) has been fully Unicode-enabled

since 2008 with the release of Delphi 2009. All strings (and string lists) managed by components are declared as string, so they now match the new UnicodeString type.

Some of the low-level, internal areas of the RTL, though, rely on different formats. For example property names are based on UTF-8, and so is part of the RTTI support available in the TypInfo unit. Beside some very specific exceptions, though, everything else has been migrated to UnicodeString and UTF-16.

Unicode support is a key element, but not the only feature that helps to build international applications.

Regarding source code files keep in mind you can save them in any format you like, but it is necessary to use a Unicode format in case you are using any code point above 255 in your source code (for identifier names, strings, comments, or just about anything else). The editor will prompt you to use such a format when required, but you can go for Unicode source files anyway.

# About The Author

This white paper has been written for Embarcadero Technologies by Marco Cantù, Product Manager for Delphi and RAD Studio, author of the best-selling series *Mastering Delphi*. The content has been extracted from his book "*Delphi 2009 Handbook*", http://www.marcocantu.com/dh2009. You can read about Marco on his blog (http://blog.marcocantu.com) and reach him at his email address marco.cantu@embarcadero.com.

# About Embarcadero Technologies

Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance, and accelerate innovation. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com.