Tech Note

embarcadero

Delphi Unicode Migration for Mere Mortals:
Stories and Advice from the Front Lines

Cary Jensen, Jensen Data Systems, Inc.

December 2009
(updated October 2010)

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
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

SUMMARY

With the release of Embarcadero® RAD Studio XE (and beginning with the release of RAD Studio 2009), Embarcadero Technologies has empowered you, the Delphi® and C++Builder® developer, to deliver first class, Unicode-enabled applications to your customers. While this important development is opening new markets for your software, in some cases it presents a challenge for existing applications and development techniques, especially where code has included assumptions about the size of strings.

This paper aims to guide your Unicode migration efforts by sharing the experiences and insights of numerous Delphi developers who have already made the journey. It begins with a general introduction of the issues, followed by a brief overview of Unicode basics. This is followed by a systematic look at the various aspects of your applications that may require attention, with examples and suggestions based on real world experience. A list of references that may aid your Unicode migration efforts can be found at the end of this paper.

INTRODUCTION

Embarcadero introduced full Unicode support in RAD Studio for the first time in August of 2008. In doing so, they ensured that Delphi and C++Builder would remain at the forefront of native application development on the Windows platform for a very long time to come.

However, unlike many of the other major enhancements that have been introduced in Delphi over the years, such as variants and interfaces (Delphi 3), frames (Delphi 5), function inlining and nested classes (Delphi 2005) and generics (Delphi 2009), enabling Unicode didn't involve simply adding new features to what was already supported in Delphi. Instead, it involved a radical change to several fundamental data types that appear in nearly every Delphi application. Specifically, the definitions for the String, Char, and PChar types changed.

These changes were not adopted lightly. Instead, they were introduced only after extensive consideration for the impact that these changes would have for existing applications as well as how they would affect future development. In addition, Embarcadero sought the input and advice of many of its Technology Partners who support and promote Delphi.

In reality, there was no way to implement the Unicode support without some inconvenience. As one of the contributors to this paper, who requested that I refer to him simply as Steve, noted, "I think PChars and Strings should never have changed meaning. ... Having said that, any choice the developers of Delphi made would have been criticized. It was a bit of a no-win situation."

In the end, changing the meaning of String, Char, and PChar was determined to be the least disruptive path, though not without consequences. On the plus side, Embarcadero instantly enabled RAD Studio developers to build world class applications that treat both the graphical interfaces and the data they help manipulate in a globally-conscious manner, removing substantial barriers to building and deploying applications in an increasingly global marketplace.

But there was a down side as well. The changes to String, Char, and PChar introduced potential problems, significant or otherwise, for the migration of applications, libraries, shared units, and time-test techniques from earlier versions of Delphi/C++Builder.

Let's be realistic about this. Nearly every upgrade of an existing application can potentially encounter migration issues that require changes to the existing code or require upgrades to newer versions of third-party component sets or libraries. The same is true when upgrading to Delphi 2009 or later. Some upgrades will be easier, and some will be more challenging.

And now we get to real point of this paper. Because of the changes to several fundamental data types, data types that we have relied upon since Delphi 1 (Char and PChar) or Delphi 2 (String), it is fair to say that migrating an existing application to Delphi 2009 or later requires more effort than any previous migration.

Contributor Roger Connell of Innova *Solutions* Pty Ltd offered this observation, "While [the Delphi team has], in my view, done a sterling job [adding Unicode support, this] has been the most challenging (in fact the only really challenging) Delphi migration." Fortunately, there are solutions for every challenge you will encounter, and this paper is here to help.

I began this project by asking the Delphi community for their input. Specifically, I asked developers who successfully migrated their existing applications to Delphi 2009 and later to share their insights, advice, and stories of Unicode migration. What I received in response was fascinating.

The developers who responded represent nearly every category of developer you can imagine. Some are independent developers while others are members of a development team. Some produce vertical market products, some build in-house applications, and some publish highly popular third-party component sets and tools used by application developers. Yet others are highly respected authorities on Delphi, developers who speak at conferences and write the books most of us have read.

Their stories, advice, and approaches were equally varied. While some described migration projects that were rather straightforward, others found the migration process difficult, especially in the cases of applications that have been around for a long time, and included a wide variety of techniques and solutions.

Regardless of whether a particular migration was smooth or challenging, a set of common approaches, practical solutions, and issues to consider emerged, and I look forward to sharing those with you.

But the story does not end with the publication of this white paper. I hope to continue to collect Unicode migration success stories, and update this paper sometime in the future. As a result, if you are inspired by what you read, and have a story of your own that complements or extends what you read here, consider becoming a contributor yourself. I'll say more about this at the end of this paper.

In the next section, I provide a brief summary of basic Unicode definitions and descriptions. If you are already familiar with Unicode, have a basic understanding of UTF-8 and UTF-16, and know the difference between code pages and code points, you should either skip this section, or quickly skim if for terms you are unfamiliar with.

But before we continue, there is one more point that I want to make. RAD Studio's support for Unicode has two complementary, though distinct, implications for those applications you build. The first is related to how strings are treated differently in code written in Delphi 2009 and later versus how they are treated in earlier versions of Delphi. The second relates to localization, the process of adapting software to the language and culture of a market.

This paper is designed specifically to address the first of these two concerns. Implementing support for multiple languages and character sets is beyond the scope of this paper, and will not be discussed further.

What is Unicode?

Unicode is a standard specification for encoding all of the characters and symbols of all of the worlds written languages for storage, retrieval, and display by digital computers. Similar to the ANSI (American National Standards Institute) code standard character set, which represents both control characters (such as tab, line feed, and form feed) and printable characters of the 26 character Latin alphabet, Unicode assigns at least one unique number to every character.

Also like the ANSI code standard, Unicode represents many types of symbols, such as those for currency, scientific and mathematical notation, and other types of exotic characters. In order to reference such a large number of symbols (there are currently more than a million), Unicode characters can require up to 4 bytes (32 bits) of data. By comparison, the ANSI code standard is based on 8-bit encoding, which limits it to 255 different characters at a time.

Each control character, character, or symbol in Unicode is assigned a numeric value, called its *code point*. The code point for a given character, once assigned by the Unicode

Technical Committee, is immutable. For example, the code point for 'A' is 65 ($0041 hex, which in Unicode notation is represented as U+0041). Each character is also assigned a unique, immutable name, which in this case is 'LATIN CAPITAL LETTER A.' Both of these can never be changed, ensuring that today's encoding can be relied upon indefinitely.

Each code point can be represented in either one, two, or four bytes, with the bulk of common code points (64K worth) being capable of being represented in two bytes or less. In Unicode terms, these first 64K symbols are referred to as the *basic multilingual plane*, or BMP (you'll want to remember these initials, as they will come up a lot in this paper).

To make things somewhat more complicated, the Unicode standard allows some characters to be represented by two or more consecutive code points. These characters are referred to as composite, or decomposable, characters.

For example, the character ö can be represented as $00F6. This character is referred to as a precomposed character. However, it can also be represented by the o character ($006F) followed by the diaeresis (¨) character ($0308). The Unicode processing rules compose these two characters together to make a single character.

This is demonstrated in the following code segment:

```
var
 s: String;
begin
  ListBox1.Items.Clear;
  s := #$00F6;
  ListBox1.Items.Add('ö');
  ListBox1.Items.Add(s);
  ListBox1.Items.Add((IntToStr(Ord('ö'))));
  s := #$006F + #$0308;
  ListBox1.Items.Add(s);
```

The purpose of composite characters is to permit a finer grain analysis of the contents of a Unicode file. For example, a researcher who wanted to count the frequency of the use of the diaeresis (¨) diacritic, regardless of which character it appeared over, could decompose all characters that use it, thereby making the counting process straightforward.

Although all currently assigned code points (as well as all imaginable future code points) can be reliably represented by four bytes, it does not make sense in all cases to represent each character with this much memory. Most English speakers, for example, use a rather small set of characters (less than 100 or so).

As a result, Unicode also specifies a number of different encoding standards for representing code points, each offering trade-offs in consistency, processing, and storage requirements. Of these, the ones that you will run into most often in Delphi are UTF-8, UTF-16, and UTF-32. (UTF stands for Unicode Transformation Format or UCS

Transformation Format, depending on who you ask.) You will also occasionally encounter UCS-2 and UCS-4 (where UCS stands for Universal Character Set).

UTF-8 stores code points with one, two, three or four bytes, depending on the size of the integer representing the code point. This is the preferred format for standards such as HTML and XML, where size matters. Specifically, characters, such as those in the Latin alphabet, which can be represented with a single byte, and which make up the bulk of HTML (at least in the majority of Web pages), use only a single byte. Only those code points that cannot be represented in 7 bits make use of additional bytes (as soon as the code point value is higher than 127, UTF-8 requires at least 2 bytes in order to encode the value). While this requires additional processing, it minimizes the amount of memory needed to represent the text, and, consequently, the amount of bandwidth required to transfer this information across a network.

UTF-16 provides something of a middle ground. For those environments where physical memory and bandwidth is less important than processing, the BMP characters are all represented in 2 bytes (16 bits) of data, which is referred to as a *code unit*. In other words, code points in the BMP are represented by a single code unit.

Earlier in this section I described how UTF-8 can use 1, 2, 3 or 4 bytes to encode a single Unicode code point. With respect to UTF-16, there is a similar, yet different situation, which occurs when your application needs to represent a character outside the BMP. These code points require two code units (4 bytes), which together form what is called a surrogate pair. UTF-16 allows you to represent code points that need more than 16 bits, by using surrogate pairs, and, together, the pair of code units uniquely identify a single code point.

UTF-32, predictably, represents all code points using four bytes. While the least economical in terms of physical storage, it requires the least processing.

In addition, UTF-16 and UTF-32 (as well as UCS-2 and UCS-4) come in two flavors: big-endian (BE) and little-endian (LE). Big-endian encoding leads with the most significant byte, while little-endian leads with the least significant byte. Which approach is used is usually identified by a byte order mark (BOM) at the beginning of an encoded file. The BOM also distinguishes between UTF-8, UTF-16, and UTF-32.

Unlike UTF-16, which can contain either 2 or 4 bytes per character, UCS-2 is always 2 bytes. As a result, it can only reference characters in the BMP. To put this another way, UCS-2 and UTF-16 are identical with respect to the BMP. However, UCS-2 does not recognize surrogate pairs, and cannot represent characters outside of the BMP.

UCS-4, by comparison, is four bytes in length, and can represent the same set of Unicode code points that UTF-32 can. The UTF-32 standard, however, defines additional Unicode features, and has effectively replaced UCS-4.

Ok, that's enough of the technical stuff. In the next section we'll see how this affects us as Delphi developers.

Unicode Migration and Delphi Applications

Unicode support in Delphi did not originate in Delphi 2009, it simply became pervasive with this release. For example, in Delphi 2007, many of the dbExpress drivers that worked with Unicode-enabled servers supported Unicode. In addition, since Delphi 2005, Delphi has been capable of saving and compiling source files in UTF-8 format. And then there's the WideString type, a two-byte string type, which has been available since Delphi 3.

In fact, one of the contributors to this paper, Steve, wrote "the biggest problem I had [with migrating to Delphi 2009] was that the application had already been made Unicode compatible using WideStrings and TNT controls. This made it harder, I guess, than an application that still used Strings and PChars."

For Delphi 2009 and later, things have changed radically. For example, component names, method names, variable names, constant names, string literals, and the like, can use Unicode strings. But for most developers, the biggest change can be found in the string and character data types. This section begins with a broad look at the changes that have been made to the string and character types. It continues with specific areas of Delphi application development that are affected by these changes.

Strings, Chars, and PChars

The String type is now defined by the UnicodeString type, which is a UTF-16 string. Similarly, Char type is now WideChar, a two-byte character type, and PChar is a PWideChar, a pointer to a two-byte Char.

The significant point about the changes to these basic data types is that, at a minimum, each character is represented by at least one code unit (two bytes), and sometimes more.

Coincidental to these changes is that the size of a string, in bytes, is no longer equal to the number of characters in the string (unless you were using a multibyte character set, like Chinese. In that case, Delphi's new Unicode implementation actually has simplified things). Likewise, a value of type Char is no longer a single byte; it is two bytes.

The old string type that you've grown to know and love, AnsiString, still exists. Just as before, AnsiString values contain one 8 byte ANSI value per character, is reference counted, and uses copy-on-write semantics. And, if you want an 8-bit character type or an 8-bit character pointer, the AnsiChar and PAnsiChar types, respectively, are also still available.

Even the traditional Pascal String still exists. These strings, which are not reference counted, can contain up to a maximum of 255 bytes. These strings are defined by the ShortString data type, contain their characters in elements 1 through 255, and maintain the length of the string in the 8-bit zeroth byte.

If you want to continue using AnsiString variables, you can. There is even a special unit, called AnsiStrings.pas, that includes AnsiString versions of many of the traditional string manipulation functions (such as UpperCase and Trim). In addition, many of the classic string-related functions are overloaded, providing you with both AnsiString and UnicodeString versions. In fact, converting existing String declarations to AnsiString declarations is an effective technique when migrating legacy code, as you will learn from a number of contributors to this paper.

Consider the following code snippet, which declares a variable s as an AnsiString:

```
var
  s: AnsiString;
…
```

What *is* different between Delphi 2009 and earlier versions, is the following declaration:

```
var
  s: String;
…
```

Here, the variable s is of type UnicodeString. While UnicodeString types share a number of features with AnsiString types, there are very significant differences. The primary similarity they share is that they are reference counted, and exhibit copy-on-write behavior.

Reference counting means that Delphi internally keeps track of what code is referring to the string. When code no longer refers to the string, memory used by the string is automatically de-allocated.

Copy-on-write is another efficiency. For those types that support copy-on-write (which in Delphi includes dynamic arrays), if you have two or more variables that refer to a given value, they all refer to the same memory location, so long as you have not attempted to change the value referred to by one of the variables. However, once you change the value referred to by one of the variables, a copy is made and the changes are applied to the copy only.

Unlike String, the WideString type is the same as when it was originally introduced in Delphi. Though it represents a two-byte character reference, it is neither reference counted nor does it support copy-on-write. It is also less efficient, performance-wise, as it does not use Delphi's FastMM memory manager. Though some developers used WideString to implement Unicode support in pre-Delphi 2009, its primary purpose was to support COM development, and mapped to the BSTR COM data type.

The AnsiString class, which behaves like pre-2009 String types, is different from its predecessors in one significant way; the internal structure is different. When memory was laid out for a traditional AnsiString, it contained one byte for each character in the string plus eight additional bytes. Four of these additional bytes held the length of the AnsiString, and the other four were used for reference counting.

By comparison, now both the AnsiString and UnicodeString types use twelve additional bytes (in addition to the memory required to hold the character data), four more bytes than the traditional AnsiStrings. Like traditional AnsiStrings, the last eight bytes are used for the string length (in characters for AnsiStrings and code units for UnicodeStrings) and reference counting. Of the additional four bytes used in both AnsiStrings and UnicodeStrings, two represent the element size of the characters, and the remaining two refer to the string's code page.

The element size of AnsiString is 1, and currently, the UnicodeString element size is 2 (but this could conceivably change in the future, which is why there is room in the internal structure). Code page, on the other hand, is a more involved topic, and is discussed later in this paper in conjunction with the issue of string conversions.

Getting Started

Let’s begin with the good news. Some legacy applications convert to Delphi 2009 and later with few or no modifications. To the extent that you are working primarily with VCL components (whose support for Unicode has in most cases been considered carefully), or components from third-party vendors who have taken the time to understand the implications of Unicode support, you have an advantage.

One of the contributors, Rej Cloutier, a programmer/software architect, reported that he has not actually made a complete conversion of his application yet, but did perform a test migration to Delphi 2010. "The result," he wrote, "was a very effortlessly migration. For one thing, all [our] string functions are encapsulated in a single unit ...[as a result], only one unit require a close lookup (about 3-4 minor modifications). About 8 DBMSs compiled successfully (each have between 100k and 185k lines of code)."

“Here is a different example. I have literally hundreds of Delphi projects that I use in my training material. Some of these projects were originally written way back in the Delphi 1 days, while others are new, demonstrating some of Delphi's newest features. Over the years, I have kept these projects up to date as I've updated my training material. As a result, most have been most recently compiled with BDS 2006 or RAD Studio 2007.”

“Since the release of Delphi 2009 I have migrated over 100 of these projects to Delphi 2009 or Delphi 2010. In all, only five or so of these projects required modifications, and

those were primarily associated with passing data to routines in DLLs and reading/writing text files."

Is this a fair representation of the ease with which legacy Delphi application can be converted? No. As contributor Steve rightly asserted, "I really do not think you can compare code samples to complicated real world applications."

But there is a lesson here. Those demonstration projects were built to show how to use features found in Delphi, and they touched on topics such as packages, DLLs, components, Delphi's open tools API, COM, DataSnap, user interface design, threads and thread synchronization, and much more. In other words, most of those applications demonstrated Delphi's RTL (runtime library), VCL (visual component library), compiler and debugger options, the integrated development environment, and Delphi's editor. And, these things did not break, for the most part. In other words, the migration of the Delphi environment to Unicode was consistent and cohesive.

It's when you reach outside of Delphi's immediate realm where things can get difficult, and which is also why Steve's observation about demonstration projects is correct. Real world applications are normally rich is features and leverage not only capabilities of the operating system directly, but also rely on outside libraries, packages, streams, files, and code. This, it turns out, is where you can run into issues.

Another difference between code demos and real world applications is that most legacy applications that are worth migrating have been around for a while. As a result, they often use techniques that were originally important for performance or features, but for which there are better alternatives today. Similarly, over time, they may have been written by different developers using somewhat different approaches. Also, third-party tools and libraries that were originally used may no longer be supported. The list goes on.

If you are looking for some kind of objective measure of the complexity of your Unicode migration, contributor Steffen Friismose suggests that you take a look at the Unicode Statistics Tool, which you can download from Embarcadero's Code Central. The Unicode Statistics Tool examines your source code, and produces an estimate of the relative complexity of your Unicode migration. You can find this tool, and its description, at http://cc.embarcadero.com/item/27398.

Based on the input I received from the many contributors to this paper, you may need to consider the following issues and techniques when migration an existing application to Delphi 2009 or later:

- The size of Strings and Chars
- Falling back to AnsiString
- String conversions

- Sets of Char
- Pointer operations and buffers
- Reading and writing external data
- External libraries and third-party components
- Database-related issues

Each of these topics will be considered separately.

THE SIZE OF STRINGS AND CHARACTERS

In the pre-Delphi 2009 days, the size of a string, in bytes, was predictable. Nowadays, it's not that simple. Because the UnicodeString type is UTF-16, you might be inclined to conclude that the size of a string, in bytes, is equal to 2 times the number of characters it contains (since a Char is two bytes long). In other words:

```
var
  SizeOfString: Integer;
  MyString: String;
begin
  MyString := 'Hello World';
  SizeOfString := Length(MyString) * 2;
```

And, yes, this works almost all of the time. And the next code segment is even better:

```
var
  SizeOfString: Integer;
  MyString: String;
begin
  MyString := 'Hello World';
  SizeOfString := Length(MyString) * StringElementSize(MyString);
```

The second example is better because (drum roll please), it makes fewer assumptions about the size of strings. Specifically, it uses the StringElementSize function to calculate the size of Char in bytes, rather than just assuming that it is the value 2.

But if you are interested in how many characters a particular string contains, it's not so simple. You might be tempted to think that the Length function returns the number of characters in a string, but that is not so. Length returns the number of code units in a UnicodeString.

The issue was best expressed by contributor Jasper Potjer of Unit 4 Agresso, who wrote, "Imagine a 5-character UTF-16 string containing [one] surrogate pair. Will Length return the number of characters [code points] (5), or the number of 16-bit words [code units] (6)?" He had several additional, related questions, and I will take the liberty here to paraphrase the essence of his overall question.

1. Does the Length function return the number of code points or the number of code units?

2. If the first character in a UnicodeString is represented by a surrogate pair, does MyString[1] contain the code point (the character) or the code unit (half of the surrogate pair)?

3. Can a Char type hold a surrogate pair? In other words, does a Char hold a code point or a code unit?

4. If the Length function returns code units, and not code points, how can I determine how many characters there are in a UnicodeString?

Oddly, in doing research for this paper, I didn't find very many discussions of this. The one exception was a blog by Jacob Thurman, which you can find at http://www.jacobthurman.com/?p=30. I also consulted with Seppy Bloom and Thom Gerdes, both of whom work on the Delphi development team.

And, once again, paraphrasing, here are the answers to the preceding questions.

1. Each element in a UnicodeString is a code unit. As a result, the size of a string, in bytes, is its length multiplied by the size of its elements (StringElementSize or 2, take your pick). While the length of a UnicodeString in characters is often the same as its length in code points, this is not true when a UnicodeString contains surrogate pairs.

2. MyString[1] contains a code unit, which may or may not be a code point.

3. No, a single Char cannot hold a surrogate pair. A Char can hold a single code unit.

4. To accurately determine the number of characters in a UnicodeString, you can use one of the helper functions in the SysUtils unit. For example, if your UnicodeString contains a mixture of BMP characters and surrogate pairs, use the ElementToCharLen function. (In short, you take an approach similar to the one needed when using multibyte character sets prior to Delphi 2009.)

These answers are demonstrated in the following code segment:

```
var
 s: String;
begin
  s := 'Look '#$D840#$DC01'!';
  ListBox1.Items.Add(s);
  ListBox1.Items.Add(IntToStr(Length(s)));
  ListBox1.Items.Add(IntToHex(Ord(s[6]),0));
  ListBox1.Items.Add(IntToHex(Ord(s[7]),0));
  ListBox1.Items.Add(IntToStr(Length(s) * StringElementSize(s)));
  ListBox1.Items.Add(IntToStr(ElementToCharLen(s, Length(s))));
```

The resulting contents of ListBox1 look like the following figure.

Look 𠂉!
8
D840
DC01
16
7

Although there are 7 characters in the printed string, the UnicodeString contains 8 code units, as returned by the Length function. Inspection of the 6th and 7th elements of the UnicodeString reveal the high and low surrogate values, each of which are code units. And, though the size of the UnicodeString is 16 bytes, ElementToCharLen accurately returns that there were a total of 7 code points in the string.

While these answers suffice for surrogate pairs, unfortunately, things are not exactly the same when it comes to composite characters. Specifically, when a UnicodeString contains at least one composite character, that composite character may occupy two or more code units, though only one actual character will appear in the displayed string. Furthermore, ElementToCharLen is designed specifically to handle surrogate pairs, and not composite characters.

Actually, composite characters introduce an issue of string normalization, which is not currently handled by Delphi's RTL (runtime library). When I asked Seppy Bloom about this, he replied that Microsoft has recently added normalization APIs (application programming interfaces) to some of the latest versions of Windows, ® including Windows® Vista, Windows® Server 2008, and Windows® 7.

Seppy was also kind enough to offer a code sample of how you might count the number of characters in a UnicodeString that includes at least one composite character. I am including this code here for your benefit, but I must offer these cautions. First, this code has not been thoroughly tested, and has not been certified. If you use it, you do so at your own risk. Second, be aware that this code will not work on pre-Windows XP installations, and will only work with Windows XP if you have installed the Microsoft Internationalized Domain Names (IDN) Mitigation APIs 1.1.

Here is the code segment:

```
const
  NormalizationOther   = 0;
  NormalizationC       = 1;
  NormalizationD       = 2;
  NormalizationKC      = 5;
  NormalizationKD      = 6;

function IsNormalizedString(NormForm: Integer; lpString: LPCWSTR;
  cwLength: Integer): BOOL; stdcall; external 'Normaliz.dll';
```

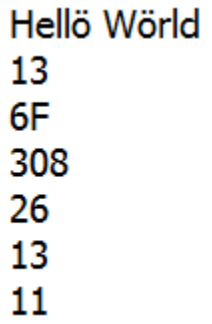
```
function NormalizeString(NormForm: Integer; lpSrcString: LPCWSTR;
  cwSrcLength: Integer; lpDstString: LPWSTR;
  cwDstLength: Integer): Integer; stdcall; external 'Normaliz.dll';

function NormalizedStringLength(const S: string): Integer;
var
  Buf: string;
begin
  if not IsNormalizedString(NormalizationC, PChar(S), -1) then
  begin
    SetLength(Buf, NormalizeString(NormalizationC,
              PChar(S), Length(S), nil, 0));
    Result := NormalizeString(NormalizationC, PChar(S),
                              Length(S), PChar(Buf), Length(Buf));
  end
  else
    Result := Length(S);
end;
```

The following code snippet, which includes a UnicodeString in which two composite characters appear, demonstrate the use of the NormalizedStringLength function:

```
var
 s: String;
begin
  ListBox1.Items.Clear;
  s := 'Hell'#$006F + #$0308' W'#$006F + #$0308'rld';
  ListBox1.Items.Add(s);
  ListBox1.Items.Add(IntToStr(Length(s)));
  ListBox1.Items.Add(IntToHex(Ord(s[5]),0));
  ListBox1.Items.Add(IntToHex(Ord(s[6]),0));
  ListBox1.Items.Add(IntToStr(Length(s) * StringElementSize(s)));
  ListBox1.Items.Add(IntToStr(ElementToCharLen(s, Length(s))));
  ListBox1.Items.Add(IntToStr(NormalizedStringLength(s)));
```

The resulting contents of ListBox1 look like the following figure:



As you can see, the displayed string contains 11 letters, though Length returns 13 code units (which is correct). Furthermore, the 5th and 6th elements of the UnicodeString contain the component parts of the first composite character. Finally, although ElementToCharLen reports that there are 13 characters, NormalizedStringLength reports that it displays 11 characters.

What should you make of this? Is ElementToCharLen incorrect? Actually, no. The UnicodeString does contain 13 code points — it's just that the Unicode rules twice combines two of these code points into a composite code point, which results in the characters that are displayed. (It normalizes the string.)

Compare this to the prior example with surrogate pairs. Each surrogate pair required two code units, but these code units represented a single code point. ElementToCharLen is counting code points.

Remember the example I mentioned earlier in this paper when introducing composite characters. I suggested that a researcher may be interested in the frequency of the diaeresis, independent of the character over which it appears. In that situation the diaeresis is a distinct character, for counting purposes. In any case, I should mention that composite characters are quite rare in normal applications, being reserved for special cases, like the researcher example.

Before leaving this section, I want to say something about the Character.pas unit, which first appeared in Delphi 2009. That unit contains the TCharacter class, along with a large number of class functions that can be used to identify information about individual characters in a UnicodeString. Each of the class functions also have a corresponding stand-alone function that calls it directly.

For example, there are functions that identify whether a particular character is an upper case or a lower case character, whether it is a symbol, a punctuation character, or a control character. There are also functions to convert individual characters to and from UTF-32.

For some reason, the Character.pas unit ended up being mentioned only one other time, and then only in passing, in this paper. Take a look at this unit. There's some nice stuff there.

FALLING BACK TO ANSISTRING

From the feedback I received from contributors, there are two general approaches to migrating existing applications to Delphi 2009 and later. One is to leave your String, Char, and PChar declarations as they are and focus your attention on those instances where these new Unicode types are invalid (such as calls that pass a PChar to an external procedure that requires a byte array).

The second approach is to convert String declarations to AnsiString, Chars to WideChars, and PChars to PWideChars. This "go with what you know" approach tends to minimize the impedance mismatch between ANSI characters and UTF-16 characters.

There is a strong argument that can be made for embracing Unicode when migrating your existing applications. Marco Cantù, author of the *Delphi 2009 Handbook*, writes, "in most cases you really want to convert your code to using the new UnicodeString type" and he

notes that there are a number of efficiencies gained by doing so, such as improving the speed of many calls to the Windows API (which is mostly Unicode these days).

But in most cases, it's not really a matter of going all Unicode versus reverting to all AnsiString, but rather a mixture of these two approaches. As Marco points out, "When 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."

But there is a purely practical side to this issue as well. If you need to get the conversion done fast, and there is little other refactoring going on, it may simply be more expedient to stay with single byte Char types. On the other hand, if your application is undergoing a major facelift, is going to be maintained actively for a long time in the future, and you have the luxury of time to make the changes, a strong argument can be made for a full boat Unicode conversion.

Let's first take a look at the AnsiString approach. Roger Connell wrote, "The [Embarcadero] supplied list of things to do [to migrate legacy code to Unicode] provided an intolerable burden with so many lines of code [needing attention]. I chose to maintain strings within my code as AnsiString and [I] put together a converter to do that. I plan to slowly retrofit ... Unicode support [later, as time permits]."

There is simply nothing wrong with this approach. And, Roger, a member of the Australian Delphi User Group (ADUG), has been so kind as to make his conversion utility publicly available. You can find this utility, along with the prerequisite cautions about its use, at:

http://www.innovasolutions.com.au/delphistuf/ADUGStringToAnsiStringConv.htm

In addition to providing a fast track to Delphi 2009 and later, Roger writes, "[This approach] leaves you with code that compiles in D6, D7 and D2009. You may get some performance hits in the UI but my logic is ... [that the performance hits] were there in D7."

However, simply falling back to AnsiStrings is not always the answer. Contributor Mariano Vincent de Urquiza of MVU Technologies LLC wrote, "A massive replacement of String to AnsiString and Char to AnsiChar didn't cut it. Every procedure had to be reviewed and tested and this spread to multiple units; this was really frustrating, I thought it would never end."

Lars Dybdahl, Software Development Manager at Daintel ApS, advocates a variety of approaches to handling strings, depending on the type of code. For example, he wrote, "We had some very old code, using pointers and external components, which was really difficult to convert. However, the amount of string data that goes in and out of this code was low, so the easy solution was to rename String to RawByteString, PChar to PAnsiChar and use the ANSI versions of the Windows API. This meant that this part of the program

does not support Unicode, but in some cases that can be ok, like encryption modules where binary data is handled in string variables."

Lars offers this additional suggestion: "After the renaming [of the data types], and making the unit work, it can often be helpful to change the interface to use String (UnicodeString), so that other units, which use UnicodeStrings, can use this unit without producing warnings. This basically encapsulates the conversion to/from UnicodeString in the implementation section."
Along these same lines, a contributor who asked to remain anonymous wrote, "I wrote a D2007 CharInSet [function] and used that where needed. I also changed Char to AnsiChar here and there (Windows API [calls], third party DLL interfaces, file format definitions, etc.). [I also] got rid of some "text" type files. When D2009 arrived I tried the demo version, and it all worked in a day or two. I've always been a bit scared of PChar, that might have helped."

While Roger Connell noted that changing your String declarations to AnsiString permitted code to be backward compatible with earlier versions of Delphi, such a conversion is not actually necessary. Some developers have had success using the String declarations just as they are (which will compile as either AnsiString or UnicodeString, depending on the version of Delphi).

Nard Moseley of Digital Metaphors, publisher of ReportBuilder, a popular reporting tool for Delphi, describes their migration: "ReportBuilder is a large complex code base of over 700,000 lines of source code. ... In moving ReportBuilder to Unicode, we followed the strategy recommended by Delphi Chief Scientist Allen Bauer. In essence, think of your application as a box. Inside the box all strings are Unicode. The outside edges of the box represent the outside edges of the application - those places where the application communicates with other systems/files/etc that may use different character encodings."

He also states: "One of the additional challenges we faced is the requirement to support the old ANSI VCL and the new Unicode VCL with a single code base. Our goal is always to minimize the amount of conditional compilation - to keep the source code as clean as possible. The strategy we focused on was to build a facade for Unicode VCL classes such as TCharacter and TEncoding. In other words, rather than calling these classes directly we call a set of internal classes that can conditionally call the TCharacter and TEncoding classes."

Another vote for the "let String be String" approach came from David Berneda, whose company Steema Software publishes TeeChart, a charting tool that both ships with Delphi and which is also available in a Professional Edition. David wrote, "We did nothing special, just making sure everything was a String (so everything compiles fine in all Delphi versions) ... when calling non-Unicode APIs, using ShortString and doing the ... PAnsiChar(text) castings."

String Conversions

String conversions occur when you assign a string of one type to that of another type. String conversions also occur when you cast a string to a data type different from the original. While string conversions are sometimes necessary in new application development with Delphi 2009 and later, they are commonplace in applications being migrated.

But before we go any further, I want to share an observation that Jan Goyvaerts wrote about in his blog. He notes that Delphi injects a lot of extra string type verification code when the $STRINGCHECKS compiler option is turned on (which is the default). He points out that, since Delphi is strongly typed, you can safely turn this compiler directive off, and gain performance benefits at the same time. C++Builder developers, however, should leave this compiler directive turned on.

When it comes to string conversions, there's good new and bad news. The good news is that string types are assignment compatible with other string types, and char types are assignment compatible with other char types. During assignment, conversion may be necessary, but in the case of string-to-string assignment, this conversion occurs automatically.

The bad news is that, depending on the conversion, there may be data loss. Understanding why this loss occurs is one of more challenging steps on the road to Unicode mastery. Let's begin by taking a deeper look at a topic mentioned in passing earlier in this paper: code pages.

Code Pages

The term code page refers to a mechanism that was used to extend the original 7-bit ASCII character set (#$00 - #$7F). In the original MS-DOS, these values in the #$80 through #$FF range were mainly used for line drawing characters. (The code pages used in MS-DOS were referred to as original equipment manufacturer, or OEM, code pages).

In earlier versions of Windows, a variety of different code pages, often called *Windows code pages*, were introduced to support the many languages that Windows needed to display. In these code pages, the characters in the #$80 through #$FF range are mostly language/culture specific characters and symbols.

Each code page is distinguished by a code page identifier. For example, most US computers use code page 1252, code page 437 refers to the OEM code page used on the original IBM PC, and code page 950 refers to the ANSI and OEM code page for Traditional Chinese.

In order to accommodate languages, such as Japanese and Chinese, where well more than 128 additional characters are needed, Windows supports both single byte code

pages as well as multibyte code pages. In multibyte code pages, one or more bytes identify a particular character. For example, in a double byte character set (DBCS), a *lead byte*, which is a value always greater than #$7F, is used in combination with a *trailing byte* to identify a particular character. Code page 932 / Japanese and code page 950 / Traditional Chinese are double byte character sets.

Each Windows installation has a default code page, and it defines the character set that is used, by default, for non-Unicode characters on that machine. It also defines the character set of the default AnsiString type (and this is true for all Delphi versions).

Earlier you learned that the layout for Strings in Delphi 2009 and later contains 2 bytes used to hold the code page. For UnicodeString types, the code page is 1200, which tells Windows that these strings are UTF-16LE. For AnsiStrings, the code page is the default Windows code page, or the code page defined for a custom AnsiString type.

But take note, by default, all AnsiString variables in an application will have the same code page, the default code page of your Windows installation. Creating two AnsiStrings with different code pages in an application is something that you have to do by explicitly telling Delphi that you want a specific code page (some examples of this are shown in the code snippets listed in the next section).

Several of the contributors to this paper also provide technical edits of one of its drafts. One of these technical editors, Lars Dybdahl, asked me to remove the reference two paragraphs back to the fact that UnicodeString has a code page. I decided to leave the reference, but I am going to quote what he told me, since I believe his reasoning will help some of you in your migration process.

Lars wrote, "I never used the code page number of UTF-16 for anything when doing Delphi 2009, or when migrating. ... one of the difficult things that I encountered, was to figure out that UnicodeString always uses the same UTF-16LE encoding. Once that was cleared up, things became much easier, because if I just kept as much as possible in UnicodeStrings, everything would just work perfectly. If the documents that I read [before I started my Unicode migration had not mentioned] the code page of UnicodeString, but focused on the fact that UnicodeStrings are easy and fast, in contrast to AnsiStrings, I would have saved valuable hours."

String Conversions and Data Loss

As mentioned previously, when converting from one string type (code page) to another, there is a possibility of data loss. Data loss will occur if the source string contains one or more characters that do not exist in the code page of the target string.

This is demonstrated with a simple example. Consider the following code snippet:

```
type
  IBMAnsi = type AnsiString(437); //IBM OEM
```

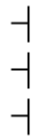
```
var
  s: IBMAnsi;
  a: AnsiString;
begin
  //Use this line if 1252 is not already your default code page
  DefaultSystemCodePage := 1252;
  s := #$B4;
  a := s;
  ListBox1.Items.Add(s);
  ListBox1.Items.Add(a);
  s := a;
  ListBox1.Items.Add(s);
```

After running this code, ListBox1 contains the following values:

The OEM code page 437 character #$B4 refers the Unicode character U+2524, which is named BOX DRAWINGS LIGHT VERTICAL AND LEFT. (It might be worth mentioning that #$B4 is an AnsiChar literal, which is not the same thing as a WideChar literal. #$2524 is a WideChar literal) This particular character does not exist in the default code page for the Windows installation on which this code was run (which in this case, is 1252). As a result, data was lost and the wrong character was printed.

The final two lines of this code segment are there to demonstrate that it's not simply a matter of the two code pages having different characters at #$B4. Here, the value of the AnsiString is passed back to the IBMAnsi variable and then displayed. As you can see, we do not get the original character back.

If we declared variable *a* to be a String (UnicodeString), instead of an AnsiString, no loss occurs. Specifically, the U+2524 character appears in both string types. This is demonstrated by the following output, which is produced when *a* is declared as String:



RawByteString

There is a special type of AnsiString called RawByteString. The RawByteString type does not have its own code page, and therefore, string assignments to a RawByteString type do not produce an implicit conversion. Instead, the code page of a value assigned to a RawByteString is the code page of whatever was assigned to it. This makes the RawByteString type an ideal data type for passing AnsiString parameters. If you pass AnsiString parameters using any other data type, an implicit type conversion will take

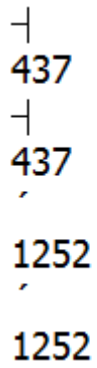
place if the two parameters do not have the same code page, possibly producing data loss.

This "adoption" of the AnsiString source code page by a RawByteString is demonstrated in the following code. Notice that when an IBMAnsi value is assigned to a RawByteString, the RawByteString adopts the code page 437. When that same RawByteString variable is assigned a value from an AnsiString that uses the default code page, it adopts the code page 1252 (which is the default code page of the Windows installation on the computer on which this code was executed).

```
type
  IBMAnsi = type AnsiString(437); //IBM OEM
var
  i: IBMAnsi;
  a: AnsiString;
  r: RawByteString;
begin
  //Use this line if 1252 is not already your default code page
  DefaultSystemCodePage := 1252;
  I := #$B4;
  r := I;
  ListBox1.Items.Add(i);
  ListBox1.Items.Add(IntToStr(StringCodePage(i)));
  ListBox1.Items.AddI;
  ListBox1.Items.Add(IntToStr(StringCodePageI));

  a := #$B4;
  r := a;
  ListBox1.Items.Add(a);
  ListBox1.Items.Add(IntToStr(StringCodePage(a)));
  ListBox1.Items.AddI;
  ListBox1.Items.Add(IntToStr(StringCodePageI));
```

After running this code, ListBox1 look like that in the following figure:



Most Unicode migrations do not have to worry about implicit conversion issues. On the other hand, if you have one of those applications that are affected by implicit code page conversions, you may be facing a more complicated Unicode conversion than most.

"Things are seriously complex," Lars Dybdahl writes, "and it easily confuses the programmer, and when the programmer does not understand how the mechanisms work, he gets frustrated about the migration process. I had to do experiments in Delphi in order to understand it, before I started the migration. Especially the fact that an ansistring(1250) variable can store a string of another code page can be annoying sometimes, because you cannot depend on the byte values inside to match the string codepage."

Lars supplied an additional code example to demonstrate some of his points, particularly about AnsiString and UnicodeString literals, RawBytesStrings, and implicit conversions. This example is quite interesting, and though longer than most of the other code samples, and I am including here to provide you with a starting point for your own experiments (I have provided a few minor edits to his comments, but otherwise preserved the original mostly the way he offered it).

```
Procedure Tform1.Button2Click(Sender: Tobject);
type
  String1250 = type AnsiString(1250);
  String1252 = type AnsiString(1252);
var
 as1:   String1250;
as1b:  String1250;
as1c:  String1250;
as2:   String1252;
s1,s2: String;
begin
DefaultSystemCodePage := 1252;
// The expressions on the right side look similar, but they are not
as1 := #$C0;                    // AnsiChar literal that has no code page
as1b := #$C0#$C0;               // UnicodeString literal
as1c := RawByteString(#$C0#$C0);// UnicodeString literal with conversion
                                //at runtime to local code page -
                                //it's not a RawByteString with $C0 values (!)
as2 := #$C0;                    // AnsiChar literal that has no code page

// Both AnsiChar literals got byte value preserved. The UnicodeString didn't.
Assert (ord(as1[1])=$C0);
Assert (ord(as1b[1])<>$C0);
Assert (ord(as1c[1])=$C0);  // as1c now has the 1252 code page
Assert (ord(as2[1])=$C0);

// Now here is a demonstration how things can be seriously confusing.
// Both as1 and as1c are String1250, but as1c now has 1252 as codepage
// because RawByteString() was used to create it. This means that both
// strings only contain $C0 values, but they don't contain the same
// characters.
Assert (length(as1)=1);
Assert (as1[1]+as1c[2] =  as1c[1]+as1c[2]);
Assert (as1   +as1c[2] <> as1c[1]+as1c[2]);

// And because of the different code pages, none of these are the same character
s1:=as1;
s2:=as2;
Assert (s1<>s2);
end;
```

EXPLICIT CONVERSIONS

So, if string conversions occur automatically, is there any need for explicit conversion? The answer is Yes. One of the more common conversions is when you need to use an AnsiString value, for example, for passing data to a Windows API call, but the data that you receive is coming from a source that is not an AnsiString.

Here is a nice example, which was contributed by well-known Delphi expert Bob Swart (popularly known as Dr. Bob). He writes, "This is one of the nicest simplified real-world examples that I have. It involves the usage of an "old" Win32 DLL exporting functions that return PChar results. The old PChar, which is now known as PAnsiChar."

We'll discuss external DLLs in more detail later, so let's concentrate more on the explicit conversion. Here is the static import statement for a simple routine in a DLL named AnsiCharDLL.dll:

```
function EchoAnsiString(const S: PAnsiChar): PAnsiChar; stdcall
  external 'AnsiCharDLL.dll';
```

As you can see, this routine takes a PAnsiChar and returns a PAnsiChar (in fact, it returns the PAnsiChar that is receives).

"We can import the function and specify it as using PAnsiChar, that's no problem", Bob writes. "However," he continues, "when calling a function [like this] that requires a PAnsiChar value, and using a TEdit (with UnicodeString value) as [the input] value, we need not one, but TWO explicit string casts."

This is demonstrated in the following code snippet:

```
ShowMessage(
  EchoAnsiString(
    PAnsiChar(AnsiString(Edit1.Text)))); // double cast !!!
```

If the value that we want to pass to EchoAnsiString is already an AnsiString, the second cast (casting to AnsiString), would not be necessary, as shown in this example:

```
var
  Msg: AnsiString;
begin
  Msg := 'Hello World';
ShowMessage(EchoAnsiString(PAnsiChar(Msg)));
```

A similar example comes from another well-known Delphi expert, Marco Cantù. This conversion, taken from page 98, uses the AnsiString cast to call GetProcAddress (a Windows API call used to dynamically get the entry point of a routine in an external DLL). Here, the name of the routine being imported is stored in strFnName, which is a UnicodeString variable:

```
GetProcAddress (hmodule, PAnsiChar (AnsiString(strFnName)));
```

Marco also suggests that you turn on all of Delphi "string conversion warnings, some of which are not enabled by default." The following list is repeated from page 88 of his book:

Explicit string cast
Explicit string cast with potential data loss
Implicit string cast
Implicit string cast with potential data loss
Narrowing given wide/Unicode string constant lost information
Narrowing given WideChar constant to AnsiChar lost information
WideChar reduced to byte char in set expression
Widening given AnsiChar constant to WideChar lost information
Widening given AnsiString constant lost information

Along these lines, contributor Lars Dybdahl suggests, "Get rid of your string warnings by fixing them, not by ignoring them. Most of them are seriously easy to fix." He also recommends, "Be careful not to create UnicodeString to AnsiString conversions that are run extremely often. For instance, like using a TStringList in an AnsiString unit, so that all assignments to/from the TStringList converts strings. This will slow down your application significantly."

For the meantime, I'll conclude this section with an interesting observation that also came from Lars, who writes, "The problem with comparing strings [is] actually very complex. For instance, let's assume that we have this code:

```
var
  line: String;
const
  myconstant: String='<something with strange unicode chars>';
...
ReadLn (file,line);
if line=myconstant then...
```

"Would this work in Delphi? Actually, I have no idea myself. I can see that the line "if line=myconstant then" compiles into a machine language call to UStrEqual, but I have no idea if this is a binary compare or a correct Unicode string comparison that handles the fact that two identical strings may use different byte values (precomposed vs. decomposed characters)."

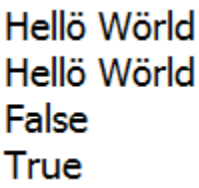
"It does not seem to use the Windows API, so my guess is that it is a binary comparison, meaning that the above code snippet would not always work ... I tried to Google an answer for this, but did not succeed. Maybe Delphi UnicodeString normally stores the normalized form of a string, but what if the UnicodeString value is read using TStream.Read? That will not normalize the string, and thus it cannot be compared byte-wise to a normalized Unicode string."

You may recall that we encountered the issue of normalization in the previous section on the size of Strings and Chars, and learned that the RTL does not directly support normalization, yet. So I think that the answer to Lars's question is that the expression line=myconstant may evaluate to False, even when the two normalized strings contain identical values.

On the other hand, there are a number of string comparison functions in the SysUtils unit that call the CompareString Windows API, and this function (actually there are two functions, an ANSI version and a Unicode version) performs the comparison based on normalized strings. The following code demonstrates the issue, as well as the solution:

```
var
 s1, s2: String;
begin
  ListBox1.Items.Clear;
  s1 := 'Hell'#$006F + #$0308' W'#$006F + #$0308'rld';
  s2 := 'Hellö Wörld';
  ListBox1.Items.Add(s1);
  ListBox1.Items.Add(s2);
  ListBox1.Items.Add(BoolToStr(s1 = s2, True));
  ListBox1.Items.Add(BoolToStr(AnsiCompareStr(s1, s2) = 0, True));
```

The contents of ListBox1 are shown in the following figure.



Sets of Char

You may recall that earlier in this paper, I quoted an anonymous contributor, one who found their conversion straightforward, as saying, "I wrote a D2007 CharInSet [function] and used that where needed." What they were alluding to is the fact that sets of Char don't really make sense any more.

The reason for this is simple. Sets in Delphi can contain, at a maximum, 256 elements, and Char is now two bytes, not one.

If you actually try to declare a set of Char, you will see the following warning:

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

You have two options here. You can either change your declaration to set of AnsiChar, or you can use CharInSet, as recommended by the compiler. Actually, CharInSet, like so

many of Delphi's String functions, is overloaded, providing you with both single byte and WideChar versions. Here is how it is declared in the SysUtils unit:

```
function CharInSet(C: AnsiChar; const CharSet: TSysCharSet): Boolean; overload;
inline;
function CharInSet(C: WideChar; const CharSet: TSysCharSet): Boolean; overload;
inline;
```

As you can see from these declarations, CharInSet returns a Boolean value if the character that you pass it is in the set you pass it. This second parameter, TSysCharSet, is declared as follows:

```
TSysCharSet = set of AnsiChar;
```

Pointer Operations and Buffers

Probably one of the single most significant and challenging aspects of Unicode migration involves code that makes use of characters in pointer operations and arrays of characters as buffers. This was reflected repeatedly by comments made by the contributors to this paper. (Just imagine what havoc will occur if an implicit string conversion is applied to a string being used as a byte array.)

For example, Olaf Monien of Delphi Experts (www.DelphiExperts.net) wrote, "[porting] code that heavily deals with PChars, buffers and pointer arithmetic ... is usually expensive as you have to revisit every single line of code." Apparently confirming this notion, an anonymous contributor who reported that their migration was rather easy, noted, "I've always been a bit scared of PChar, that might have helped."

Lars Dybdahl echoed the issue of code complexity when he wrote, "We had some very old code, using pointers and external components, which was really difficult to convert." This type of code has to be examined line-by-line.

If you don't use pointers or buffers a whole lot (and I am one of these people), you might wonder why developers use these constructs in the first place. The answer is speed and features.

Many of the people who use these techniques are either doing complex operations where speed is essential, are performing tasks for which there is (or was) no alternative, or are using techniques developed in the early days of Delphi (or Turbo Pascal). (Even when new techniques or classes arise to replace some of this old-style coding, some developers are likely to continue using these techniques out of force of habit, even at the expense of maintainability. This is just a personal observation, and not a value judgment.)

Actually, the complexity of this type of code is not related to pointers and buffers per se. The problem is due to Chars being used as pointers. So, now that the size of Strings and

Chars in bytes has changed, one of the fundamental assumptions that much of this code embraces is no longer valid: That individual Chars are one byte in length.

Since this type of code is so problematic for Unicode conversion (and maintenance in general), and will require detailed examination, a good argument can be made for refactoring this code where possible. In short, remove the Char types from these operations, and switch to another, more appropriate data type. For example, Olaf Monien wrote, "I wouldn't recommend using byte oriented operations on Char (or String) types. If you need a byte-buffer, then use 'Byte' as [the] data type: buffer: array[0..255] of Byte;."

For example, in the past you might have done something like this:

```
var
  Buffer: array[0..255] of AnsiChar;
begin
  FillChar(Buffer, Length(Buffer), 0);
```

If you merely want to convert to Unicode, you might make the following change:

```
var
  Buffer: array[0..255] of Char;
begin
  FillChar(Buffer, Length(buffer) * StringElementSize(Buffer), 0);
```

On the other hand, a good argument could be made for dropping the use of an array of Char as your buffer, and switch to an array of Byte, as Olaf suggests. This may look like this (which is similar to the first segment, but not identical to the second, due to the size of the buffer):

```
var
  Buffer: array[0..255] of Byte;
begin
  FillChar(Buffer, Length(buffer) * StringElementSize(Buffer), 0);
```

Or, alternatively:

```
var
  Buffer: array[0..255] of Byte;
begin
  FillChar(Buffer, Length(buffer) * SizeOf(Buffer), 0);
```

The advantage of these last two examples is that you have what you really wanted in the first place, a buffer that can hold byte-sized values. (And Delphi will not try to apply any form of implicit string conversion since it's working with bytes and not code units.) And, if you want to do pointer math, you can use PByte. PByte is a pointer to a Byte.

The one place where changes like may not be possible is when you are interfacing with an external library that expects a pointer to a character or character array. In those cases, they really are asking for a buffer of characters, and these are normally AnsiChar types.

In addition to using arrays of Byte, you should consider the TBytes type, which is a dynamic array of Byte. TBytes is declared in the SysUtils unit, and looks like this:

```
TBytes = array of Byte;
```

Before leaving this subject, it seems appropriate to share a few words from Allen Bauer, Chief Scientist at Embarcadero Technologies. In his blog (http://blogs.embarcadero.com/abauer/2008/01/24/38852), he wrote, "Because [we could do pointer math with] PChar ... many developers (ourselves included) would do crazy things such as casting a pointer of one type to a PChar and then do some pointer arithmetic. ...What this has done is created cases where some code is littered either with a lot of pointers cast to PChars or the direct use of PChar pointers even when the data being manipulated isn't specifically byte-sized characters. In other words, PChars were used to manipulate byte-buffers of arbitrary data."

"During the development of [RAD Studio 2009], we discovered some of our own code was doing a lot of the above things. (I told you we've all done it!) Using the PChar trick was simply the only thing available and made a lot of the code simpler and a little easier to read. ...In looking at the code, it was clear that the intent was to access this data buffer as an array of bytes, and was merely using a PChar as a convenience for accessing as an array or performing simple arithmetic."

"If you declare a typed pointer while [the $POINTERMATH compiler directive] is on, any variable of that type will allow all the scaled pointer arithmetic and array indexing you like. ...PByte is declared with that directive on. This means that all the places that are using the PChar type simply for the byte-pointer and byte-array access, can now use the PByte type instead and none of the existing code statements and logic needs to change. A simple search and replace over the source is what is needed. Sure, we could have changed the PChar references to PAnsiChar, but that simply serves to perpetuate the lack of clarity over what the intent of the code actually is."

Reading and Writing External Data

External files and streams are other areas that require attention during Unicode migration. Ray Konopka of Raize Software, a company that makes award-winning components and tools for Delphi developers, puts the issue in perspective. And although he is talking about SaveToFile and LoadFromFile for list controls, his comments apply to many situations where files or streams are written to or read from.

File IO and Text Encoding

"Most list controls provide a SaveToFile method and a LoadFromFile method," Ray writes. "These methods will typically compile under RAD Studio 2009 without any problems at all. Even at runtime, the methods will appear to work correctly. That is, until you actually put a Unicode character in one of the items."

"Calling SaveToFile will even appear to work correctly, but by default files created using SaveToFile are ANSI encoded and that means that any Unicode characters will be stored incorrectly in the file. Calling LoadFromFile to populate the list from the previously saved file will not work correctly because the actual Unicode character data will be lost."

Ray continues, "The solution, of course, is to specify the encoding that is used for the text file. To do this, the component author needs to provide overloaded versions of SaveToFile and LoadFromFile (as well as SaveToStream and LoadFromStream) that take an Encoding parameter."

"This solution is okay, but it does require that the developer using the component pick an appropriate encoding. A developer could pick a Unicode based encoding such as UTF-8 for all files and be done with it. But this would mean that even lists that contain just ANSI-based characters would get stored in a UTF8 file, which is not really necessary. What would really be nice is to save the file using UTF-8 encoding [or some other encoding] only when it was needed."

What Ray is referring to here is that almost all of the SaveToFile, LoadFromFile, SaveToStream, and LoadFromStream calls (and other similar calls) now accept an encoding as an optional second parameter. If you do not specifically define an encoding, the default encoding will be used.

You define an encoding using either properties or class functions of the TEncoding class, which appears in the SysUtils unit. Examples of TEncoding classes that are available to you include ASCII, UTF8, and Unicode.

The need to control the encoding of a file or stream was echoed by one of the anonymous contributors. "We save all of our configuration information in text streams and when compiled in 2009, the Unicode [encoding] increased the file size from 90k to 130k," they wrote. "We noticed that the TChart text (a TeeChart class) was saved using single byte characters so we [encoded] the multi-byte characters to single bytes" which saved the extra disk space.
Ray Konopka went even further, saying, "We did not want to always store the text files as UTF-8 files. Instead, we wanted to handle the files much like the Delphi IDE. That is, if a unit contains a Unicode character, then the file is saved as a UTF-8 file. However, if the contents of the unit just contain ANSI characters then the file is stored using the default encoding."

Here is the code sample that Ray contributed to demonstrate this approach:

```
{$IFDEF UNICODE}
    UseANSI := lstPreview.Items.Text =
                UnicodeString( AnsiString(lstPreview.Items.Text ) );

    if UseANSI then
```

```
        lstPreview.SaveToFile( dlgSave.FileName, TEncoding.Default )
      else
        lstPreview.SaveToFile( dlgSave.FileName, TEncoding.UTF8 );

{$ELSE}
    lstPreview.SaveToFile( dlgSave.FileName );
{$ENDIF}
```

Ray explains, "If a list contains Unicode characters, then converting the Items.Text to an AnsiString and back to a UnicodeString will [return a string] different than the original Unicode string, which means that we need to encode the file with UTF-8. If the string conversion results in no loss of data, then the strings will match and the file can be saved as ANSI."

Embarcadero has published a list of the IO routines that can accept a TEncoding reference, permitting you to specify the encoding to use. This can be located at

http://docwiki.embarcadero.com/RADStudio/en/Using_TEncoding_for_Unicode_Files

J.D. Mullin, R&D Project Manager for the Advantage Database Server (published by Sybase iAnywhere) was also kind enough to share his technique for restoring previously persisted information. Like the anonymous contributor who wanted to save a file in ANSI format, J.D. wanted to ensure that ANSI data that was previously saved was restored correctly. "Reading ANSI string data from a stream into a string buffer will not work," he writes. "You need to explicitly read [the data] into a temporary ANSI buffer first."

Here is his code sample that demonstrates this technique:

```
function SReadString(S: TStream): String;
var
  sLen: LongInt;
  temp: AnsiString;
begin
  sLen := SReadLongint(S);
  SetLength( temp, sLen );
  s.ReadBuffer( temp[1], sLen);
  result := temp;
end;
```

J.D. explains, "The reason the temporary ANSI buffer is necessary is because ReadBuffer automatically determines the type of the destination buffer and acts accordingly. If you had stored ANSI data in the file or stream, you need to read it into an ANSI buffer, not a Unicode buffer. If you pass a Unicode buffer to ReadBuffer, it is going to think you stored Unicode data and will read it out as such."

Lars Dybdahl also had insight into reading and writing files. He wrote, "Many of our existing I/O routines were designed to handle UTF-8 encoding in Delphi 2007, and this means that a lot of the logic and data storage was about manipulating UTF-8 in AnsiStrings."

"The solution was to remove all UTF-8 conversion inside the algorithm, and just apply it at the I/O point, so that all text handling used UnicodeString. For instance, TStringList worked well with UTF-8 in Delphi 2007, but in Delphi 2009 it uses UnicodeString. You should convert UTF-8 to UnicodeString as soon as possible, and definitely before putting it into a TStringList."

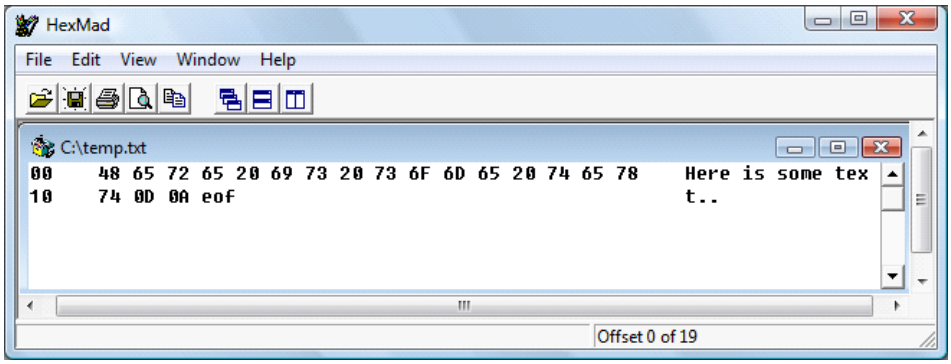
There are two more techniques that we need to discuss before concluding this section on reading and writing data. But before we do that, there is a topic, mentioned only briefly earlier in this paper, that needs additional explanation. That topic is byte order mark, or BOM.

THE BYTE ORDER MARK

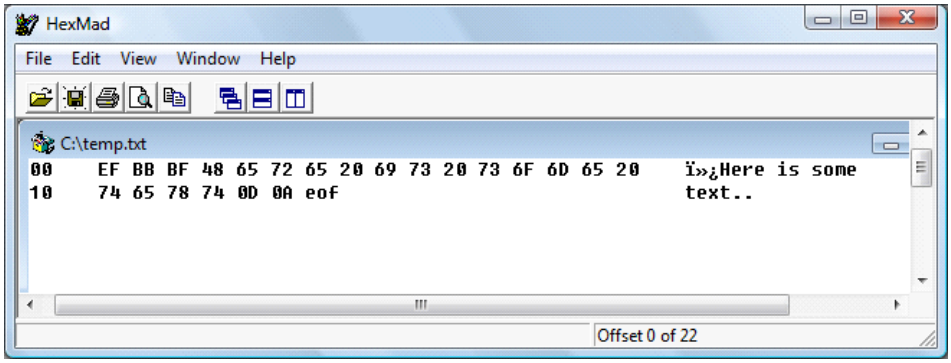
Byte order mark is a preamble that may appear in text files, and, when present, it serves to identify the file's encoding. If you use Delphi's SaveToFile methods (or similar methods where encodings can be specified, Delphi will write a BOM, if appropriate, as the first couple of bytes of the file. This can be demonstrated with the following code sample:

```
procedure TForm1.SaveWithEncodingClick(Sender: TObject);
var
  sl: TStringList;
begin
  sl := TStringList.Create;
  try
    sl.Text := TextEdit.Text;
    ListBox1.Items.Clear;
    ListBox1.Items.AddStrings(sl);
    if EncodingComboBox.Items[EncodingComboBox.ItemIndex] = 'ASCII' then
      sl.SaveToFile('c:\temp.txt', TEncoding.ASCII)
    else
    if EncodingComboBox.Items[EncodingComboBox.ItemIndex] = 'UTF-8' then
      sl.SaveToFile('c:\temp.txt', TEncoding.UTF8)
    else
    if EncodingComboBox.Items[EncodingComboBox.ItemIndex] =
      'UTF-16 LE (Little-endian)' then
      sl.SaveToFile('c:\temp.txt', TEncoding.Unicode)
    else
    if EncodingComboBox.Items[EncodingComboBox.ItemIndex] =
      'UTF-16 BE (Big-endian)' then
      sl.SaveToFile('c:\temp.txt', TEncoding.BigEndianUnicode);
  finally
    sl.Free;
  end;
end;
```

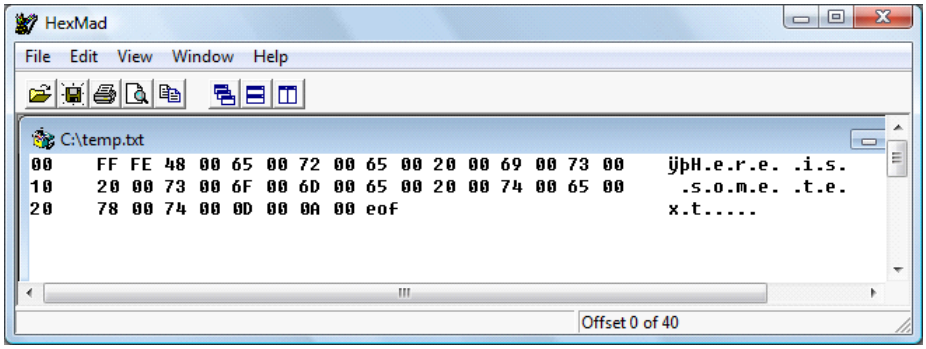
When the file is viewed by a low-level hex file viewer (HexMad, in this case) with the ASCII encoding, the file looks like that shown here:



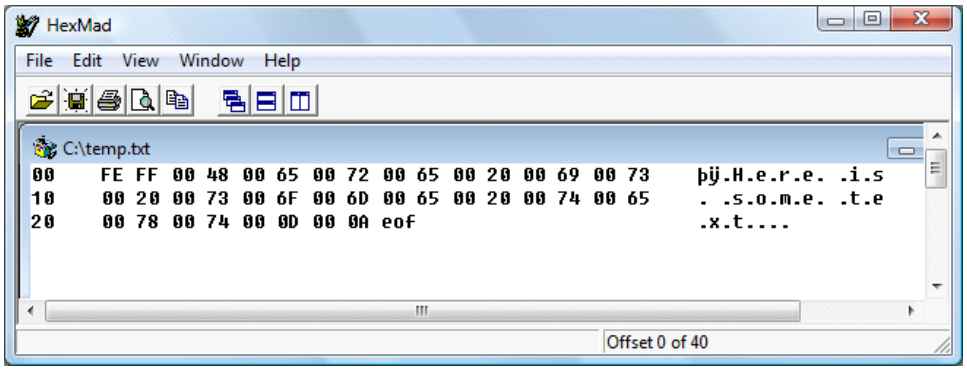
With the UTF-8 encoding, the file looks like the following:



The EF BB BF is the byte order mark (BOM), and in this case, it identifies the file as a UTF-8 Unicode file. By comparison, when the Unicode encoding (UTF-16) is selected (which defaults to little-endian), the file looks like the following. (You can read the BOM using the TEncoding.GetPreamble method.):



And it looks like this with UTF-16 BE (big-endian) selected:



You'll notice that the Unicode (UTF-16) preambles are only two bytes in length. With the little-endian (the default), the preamble is FF FE, and with big-endian, it is reversed (FE FF).

The BOM Is Optional

On page 91 of his book *Delphi 2009 Handbook*, Marco Cantù writes, "The number one recommendation, whenever saving to a file, is to save the BOM, to make it clear in which format it is. This is no more difficult to achieve when working in memory, because even if you don't remember the actual format, Delphi's streaming code adds the proper BOM even to a memory stream."

As you saw earlier in this chapter, the act of defining an encoding is sufficient to ensure that the proper BOM is written. When reading the file or stream, he continues, it is not necessary to specify an encoding, since Delphi will infer it from the BOM that was created when the data was written. He notes that it is particularly important not to specify an encoding when reading a file that you created with a specific encoding, since Delphi will let you write data using one encoding, and then not raise an exception if you attempt to read that data with a different encoding (but the data will likely be incorrect).

Things can get a little more messy because BOMs are not a prerequisite for Unicode data files. As Lars Dybdahl points out, "Many tools and applications cannot read files that include a BOM. A good example is when your application creates a configuration file for a Linux server application that will fail when reading the BOM. You may even experience XML readers that can only read UTF-8 encoded XML files if they do not have a BOM. Similarly, if you are writing text into a binary structure, like a blob field in a database, the reader may not understand a BOM."

As a result, you might encounter a file created by some other source that includes no BOM, even though it is a form of Unicode. This potentially thorny issue was also noted by contributors J.D. Mullin and Louis Kessler. Louis posted the following direct question on Stack Overflow: "How can I best guess the encoding when the bom (byte order mark) is missing?" The answers he received led him to CharSet Detector, an open source module by Nikolaj Yakowlew that examines the patterns of characters in a file and predicts the encoding. Though I haven't personally tested CharSet Detector, if you need something like this you can investigate it further at http://chsdet.sourceforge.net/.

In concluding this section on reading and writing data, I would be remiss if I didn't at least mention the availability of a pair of new stream reading and writing classes in Delphi 2009 and later. The TStreamWriter and TStreamReader classes are functionally identical to their .NET counterparts. Importantly, both permit you to define the encoding that you want to use when writing to or reading from a stream.

Using External Libraries and Third-Party Components

As the preceding discussions have highlighted, there are many different areas that may require attention when migrating your own source code. Fortunately, you presumably enjoy a number of advantages that will help you approach this task. You are likely intimately familiar with the code, it is written in either your style or that defined by your company's policy, and you have access to all of the source code.

When it comes to external libraries and third-party components, however, you may have none of these benefits. You may not even have the source code, leaving you at the mercy of forces beyond your control.

I am specifically going to address three distinct areas related to code outside of your control in this section. I am going to begin with a look at the Windows API. Next, I'll discuss third-party components. The final section will discuss external libraries for which you do have source code, those that were written by you and/or your team, or that you obtained from an open source resource.

Windows API

It's a given that your code does not operate in a vacuum. It always relies on other code "out there." At a minimum, this includes the operating system and the modules that make it work. With respect to Windows, we call these libraries the Windows API, or application programming interface.

As far as Unicode support goes, the news is good. Windows, at least since Windows 2000, has fully supported Unicode, and even earlier versions were aware of multi-byte characters. In addition, most of the Windows API associated with strings is implemented with at least two versions of each routine, one for strings that use Windows code pages (ANSI) and one for wide strings. This is immediately apparent if you examine an import unit such as windows.pas, which imports both A and W versions of the string-related routines.

Here is a typical example of a set of external declarations in the windows unit:

```
function GetModuleFileName; external kernel32 name 'GetModuleFileNameW';
function GetModuleFileNameA; external kernel32 name 'GetModuleFileNameA';
function GetModuleFileNameW; external kernel32 name 'GetModuleFileNameW';
```

What this is saying is that the GetModuleFileName, a function that will return the fully qualified name of an executable (regardless of whether it is a console application, windows application, or DLL), has three versions. There is the GetModuleFileNameA (ANSI) version, which takes a ANSI character buffer (in which the path is stored), the GetModuleFileNameW (wide) version, which takes a UTF-16 character buffer, and GetModuleFileName, an alias for the W version. As you can see in the above code snippet, the GetModuleFileName version actually calls the W version.

What is interesting to note is that in pre-Delphi 2009 versions of windows.pas, you would find the following declaration:

```
function GetModuleFileName; external kernel32 name 'GetModuleFileNameA';
function GetModuleFileNameA; external kernel32 name 'GetModuleFileNameA';
function GetModuleFileNameW; external kernel32 name 'GetModuleFileNameW';
```

So, what this is really saying is that we've had access to both the ANSI and multi-byte version of most of the string-related function calls for some time, the difference being that the wide version is now the default.

In general, this means that when there is both an ANSI and a wide version of a Windows API call, and you are using the native String, Char, and PChar types (Unicode enabled), your existing code should migrate just fine. Consider the following code sample:

```
var
  Path: array [0..255] of Char;
begin
  GetModuleFileName(HInstance, Path, Sizeof(Path));
  Label1.Caption := Path;
```

This code works, and compiles, both in Delphi 2009 and later versions, as well as in prior versions of Delphi. With Delphi 2009 and later, your code is calling the wide version, and in earlier versions you are getting the ANSI call.

On the other hand, if the necessities of your conversion lead you to take the AnsiString route (replacing String declarations with AnsiString, Char with AnsiChar, and so on), you will need to examine your Windows API calls and change them over to explicitly call the ANSI versions. For example:

```
var
  Path: array [0..255] of AnsiChar;
begin
  GetModuleFileNameA(HInstance, Path, Sizeof(Path));
  Label1.Caption := Path;
```

In those cases where a suitable ANSI version of the Windows API call does not exist, you will have to take a different approach. For example, converting or casting your incompatible data types to something supported by the available routines. Frankly,

however, I am not aware of any particular Windows API calls where such an adjustment is necessary.

THIRD-PARTY TOOLS

Unlike the Windows API, which is integral to our applications, third-party components are a convenience that we embrace to save development time and improve our application's features. And they are a double-edged sword.

When we rely on a third-party component to provide a significant amount of functionality for an application, we are entering into a relationship with that third-party vendor. When it comes time for us to migrate our application to a later version of Delphi, our application's migration must go hand-in-hand with that of the vendor.

Fortunately, there are some very talented and reliable vendors in the Delphi third-party components space. For example, four of the contributors to this paper, David Berneda (Steema Software), Ray Konopka (Raize Software), J.D. Mullin (Sybase iAnywhere), and Nard Moseley (Digital Metaphors) have not only updated their company's offerings to support Delphi 2009 and later, but they have done so in a way that maintains compatibility with earlier versions of Delphi. This is why these companies have continued to succeed and earn the devotion of their loyal customers.

On the other hand, there are no guarantees. Over the years, some third-party vendors have disappeared from the scene. And if your application relied on the components of one of these vendors, the results can range from inconvenient to disastrous.

I know many developers who will not commit to a third-party component set unless they have the option to purchase the source code, and that source code is provided in such as way that the developer can continue using that source code in the event that the third-party vendor is no longer able to support the product.

Having source code is not always the solution, but it is a darn good start. On the other hand, if no source code is available, you are left with only two alternatives. You can remove the components from your application, or you can resign yourself to continuing to maintain the application in the most current version of Delphi those components supported. Removing components from an application is painful, to say the least. Getting stuck permanently in an older version of your compiler is horrible, and is usually only acceptable if the application itself is nearing the end of its lifecycle.

NON-COMMERCIAL EXTERNAL LIBRARIES

The final set of libraries that I'll address are those libraries for which you have source code, but which are not commercially supported. These include libraries that were created by you or your team, as well as open source libraries.

With respect to custom libraries, those created in-house, the migration issues are very similar to those of any other application, with an additional twist. You not only need to address the Unicode support internal to the library, but manage the API as well. For example, you may want to introduce both ANSI and wide versions of exported functions that need to pass String and Char data in parameters.

For open source libraries, the issues are the same, yet with one more concern. In most cases, these libraries are covered under a GNU public license, or some similar agreement. Before you start investing time in migrating a non-commercial library that you do not own the rights to, make sure that you read the license agreement that accompanied it, and verify that you can abide by whatever terms that license dictates. (Contributor Steffen Friismose suggests that, if the license is a GNU public license, or something similar to it, you can often get around licensing issues by contributing you migration work to the project.)

Database-related Issues

Database applications that do not need to store or display Unicode strings typically migrate to Unicode-enabled Delphi with ease. For those applications that must handle Unicode strings, things are a bit more complicated, as you will learn a little later in this section.

But to start, let's begin with a change that is not specifically Unicode related, but one that nonetheless you might encounter as you migrate an older database application to Delphi 2009 and later. This issue is related to bookmarks.

A bookmark is a reference to a record location in a TDataSet, and it is used for navigation (using a bookmark to identify a record to which you might want to return later). There are two bookmark issues introduced in Delphi 2009. The string-based bookmark (TBookmarkStr) is deprecated, and should not be used. The second is that the TBookmark type has changed.

J.D. Mullin explains it this way: "[Embarcadero] has changed the TBookmark type from a pointer to TBytes. This will not affect most applications that simply use the GetBookmark[, GotoBookmark, and] FreeBookmark TDataSet methods. If, however, you are doing anything "goofy" with the pointer you get from GetBookmark, beware. Many of our automated tests needed to be modified to consume the bookmarks in a more generic/standard fashion."

I'm not entirely sure what kind of "goofy" things you might be inclined to do with a TBookmark. Nonetheless, J.D.'s point is well taken. A bookmark is intended to mark a record position in a TDataSet, a position to which you can quickly return by calling GotoBookmark. If you are using it for some other purpose, test your code carefully.

Turning our attention back to Unicode migration, as it relates to database applications, unfortunately things get a bit difficult. You may recall that early in this paper, I quoted a contributor who wrote, "The biggest problem I had [with migrating to Delphi 2009] was that the application had already been made Unicode compatible using WideStrings." Apparently this complexity is not unique.

In his book *Delphi 2009 Handbook*, Marco Cantù notes that pre-Delphi 2009 Unicode support in the TDataSet and TFields classes was provided using the TWideString type. Since Delphi 2009 the multi-byte string types have been declared as Strings (or even explicitly as UnicodeStrings).
While this update makes the reading and writing of Unicode data in the TField classes consistent with the UnicodeString data type, and eliminates some potential data conversion issues, some of the class and member names in the TDataSet arena remain confusing. For example, there is no TUnicodeStringField type, and the TStringField class still stores its value as an AnsiString value. If you want a Unicode TField, you use TWideStringField (which, as mentioned in the preceding paragraph, is stored as a UnicodeString in Delphi 2009 and later).

Much of this will not affect the typical database application. For example, if your database wasn't Unicode-enabled prior to Delphi 2009, you did not need the WideString classes, such as TWideStringField, and therefore will not need to accommodate their conversion to UnicodeStrings.

On the other hand, if you implemented Unicode support for your database prior to Delphi 2009, you will have to examine how you are using your WideString types, and ensure that you are doing so consistently with those UnicodeString definitions that appear in Delphi 2009 and later.

For example, Marco notes that the TDataSet.GetFieldNames method in Delphi 2006 and 2007 returned a TWideStrings value (*Delphi 2009 Handbook*, page 333). If you called that method, and assigned the value to a TWideStrings variable, a WideString to UnicodeString type conversion will now occur during that assignment if you compile your application in Dephi 2009 or later. He recommends that you "rewrite your code, locating any occurrence of the TWideStrings and TWideStringList classes and moving them to the preferred TStrings and TStringList types."

It is interesting that there were very few comments from the contributors about database migration. I hope that this means that most Unicode migrations of database applications encounter few issues related to the database side of things. In fact, the final two comments about database Unicode migration come from Lars Dybdahl.

One of these is a recommendation. Lars recommends, "Make sure that your database tools understand Unicode before migrating your database fields to Unicode." In other

words, if your tools do not understand Unicode, you should not start putting Unicode data into the database.

Preventing the injection of Unicode characters into a non-Unicode database (which would produce a data loss) may require a little bit of preventative planning. Specifically, what if you are permitting your users to edit data using data aware controls? Most of these controls support WideString data.

Here is what I suggest. Once your application is running in Delphi 2009 or later, see what happens when you intentionally add a Unicode character to your database through your user interface. Furthermore, verify what happens when you try to do something with that data, such as display it in a report.

Let's face it, we cannot always validate a user's data entry, and it may turn out that a user entering Unicode data is no less of a problem that a user entering, say, the wrong date. Garbage in, garbage out.

On the other hand, if the introduction of Unicode data into your application causes unacceptable behavior, such as access violations, you may have to change how you collect your data, verifying that it is valid before actually inserting it into your database. For example, you might need to use a ClientDataSet as an intermediate layer between your user interface and your underlying database. ClientDataSets support TWideStringFields, which can cache your user's data while it awaits validation.

When you are done collecting data, and before writing the data in the ClientDataSet to your underlying database, you could test your string-related fields for valid data. You could even use a trick similar to the one that Ray Konopka shared earlier in this paper, where you convert the TWideStringField data to AnsiString and back, verifying that no data loss occurred in the conversion, in which case, the data is ready to write to your database.

Finally, Lars shares one of the migration challenges specific to data access that his team encountered as a direct result of Delphi's Unicode support. Lars wrote: "Doing Firebird Unicode with IBX meant that a patch needed to be applied to IBX, but the hardest part was blobs. Sometimes they contain binary stuff, and sometimes they contain text, and with Delphi 2007, it really doesn't matter and [the] AsString [method] was suitable for both ...”

"But now, they need to be treated completely separately, in order to get the text fields as UTF-16 UnicodeString, and the binary stuff as RawByteString. The solution was to duplicate a lot of blob-related procedures, one for RawByteString and one for String, and then use the appropriate procedure on the appropriate fields - and we also had to introduce small functions that get the blobs correctly out of IBX."

While Lars’s observation was specific to Firebird and the IBX driver his team was using, it offers an interesting lesson in Unicode migration. As we learned in the preceding section,

when you are relying upon code outside the realm of your application, you may encounter problems not of your own doing. Nonetheless, these deficiencies need to be addressed if your migration is going to be successful.

Conclusions

Nard Moseley wrote, "Moving to Unicode seems scary at first - the unknown always is. And moving to Unicode did require us to learn some new concepts - always painful. But the fact is that most source code requires little to no modifications. Even areas such as database access and many Windows API calls will just work. By following the recommendations of the Delphi team and using tools such as the Delphi compiler warnings and the search facility, moving to Unicode can be a painless straight forward process."

As I reflect on the input of the many contributors to this paper, I have to caution that not all Unicode migrations will go as easily as Nard's quote suggests. It all depends on the complexity of the application, what it does, what it interfaces with, to what extend previous Unicode migrations have been applied (using the older WideString type), among other issues.

But the important point is that, whether your application is one of the easy conversions or one of the challenging conversions, when you are done, you will have extended the life of your application significantly. Not only will you be able to provide you application with an updated look and feel, such as support for Windows 7 features, but you will have readied your application for future enhancements planned for Delphi, such as cross platform compilation and 64-bit native code.

Acknowledgments

This paper was truly a work inspired, supported, and realized by many. I am deeply grateful to each and every individual who helped make it a reality. From Embarcadero Technologies, I want to thank Mike Rozlog, Senior Director of Delphi Solutions, for proposing that I write a Unicode migration paper, and Tim Del Chiaro, who worked diligently to get things done. I also want to thank Seppy Bloom and Thom Gerdes, Delphi team members at Embarcadero Technologies, for providing technical assistance and contributions during the writing of this paper.

I am also indebted to the many contributors who shared details about their Unicode efforts, as well as support and guidance in developing this paper. In alphabetical order, these contributors are: David Berneda, Marco Cantù, Rej Cloutier, Roger Connell, Mariano Vincent de Urquiza, Lars Dybdahl, Steffen Friismose, Louis Kessler, Ray Konopka, Olaf Monien, Nard Moseley, J.D. Mullin, Jasper Potjer, Steve, Bob Swart, Jacob Thurman, as well as several people who wished to remain anonymous.

I also want to thank Loy Anderson of Jensen Data Systems for helping to proofread and copy edit this paper. Last, but not least, I want to thank Lars Dybdahl, Steffen Friismose, Takeshi Arisawa of Embarcadero Technologies, for their insightful technical reviews of drafts of this paper.

REFERENCES

The following are a variety of resources that you may benefit from looking at. I regret that I did not create a complete list of every blog, paper, or posting that I read concerning Unicode. Fortunately, many of the more memorable are listed here.

BLOGS AND PAPERS COVERING UNICODE, DELPHI, AND SOFTWARE ISSUES

http://blogs.embarcadero.com/abauer/2008/01/28/38853
http://blogs.embarcadero.com/abauer/2008/01/10/38847
http://blogs.embarcaderor.com/abauer/2008/01/09/38845
http://www.micro-isv.asia/2009/03/make-sure-your-web-site-is-always-displayed-with-the-right-characters/
http://www.micro-isv.asia/2009/03/why-not-use-utf-8-for-everything/
http://www.micro-isv.asia/2008/12/choose-the-right-file-format-for-your-delphi-source-code/
http://www.micro-isv.asia/2008/10/delphi-2009-string-performance-in-a-nutshell/
http://www.micro-isv.asia/2008/10/needless-string-checks-with-ensureunicodestring/
http://www.micro-isv.asia/2008/09/speed-benefits-of-using-the-native-win32-string-type/
http://jdmullin.blogspot.com/2008/09/tips-when-porting-delphi-application-to_16.html
http://www.bobswart.nl/Weblog/Blogs.aspx?RootId=2:2947
Delphi in a Unicode World Part I: What is Unicode, Why do you need it, and How do you work with it in Delphi? by Nick Hodge
http://edn.embarcadero.com/article/38437
Delphi in a Unicode World Part II: New RTL Features and Classes to Support Unicode by Nick Hodge
http://edn.embarcadero.com/article/38498
Delphi in a Unicode World Part III: Unicodifying Your Code by Nick Hodge
http://edn.embarcadero.com/article/38693
Delphi and Unicode (a White Paper) by Marco Cantù
http://www.embarcadero.com/images/dm/technical-papers/delphi-and-unicode-marco-cantu.pdf
http://thedorictemple.blogspot.com
http://www.beholdgenealogy.com/blog

MICROSOFT WEB PAGE THAT DISCUSSES CODE PAGES

http://msdn.microsoft.com/en-us/library/dd317752(VS.85).aspx

The Unicode Consortium

http://unicode.org/

Books

Delphi 2009 Handbook (2009, Marco Cantù), available from Amazon.com and Lulu.com.
Delphi 2009 Development Essentials (2009, Bob Swart), available from Lulu.com

About the Author

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based company that provides software training, development, consulting, and mentoring. He is an award-winning, best-selling co-author of twenty books, including books on Advantage Database Server, Delphi, Kylix, Oracle JDeveloper, JBuilder, and Paradox. He is a popular speaker at conferences, workshops, and training seminars throughout North America and Europe. Cary has a Ph.D. in Human Factors Psychology from Rice University, specializing in human-computer interaction.

Comments and Contributions

It is my sincere wish to update this paper in this future. I warmly welcome comments, corrections, and contributions, which may be used to improve or expand this paper in the future. If you have something you would like to offer, please email it to me at cjensen@jensendatasystems.com. Please use the subject line "Unicode migration." I will reply to any email I receive within one week. If you do not receive a reply, your email slipped through the cracks. Please resend.

Embarcadero Technologies, Inc. is the leading provider of software tools that empower application developers and data management professionals to design, build, and run applications and databases more efficiently in heterogeneous IT environments. Over 90 of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero's award-winning products to optimize costs, streamline compliance, and accelerate development and innovation. Founded in 1993, Embarcadero is headquartered in San Francisco with offices located around the world. Embarcadero is online at www.embarcadero.com.