Unicode Support in Lazarus

| **English (en)** | 日本語 (ja) | русский (ru) |

Introduction

This page covers Unicode support in Lazarus **programs** (console or server, no GUI) and **applications** (GUI with LCL) using features of FPC 3.0+.

The solution is cross-platform and uses UTF-8 encoding which is different from Delphi's UTF-16, but you can write code that is fully compatible with Delphi at source code level by remembering just a few rules.

The Unicode support is enabled automatically for LCL applications since Lazarus 1.6.0 when compiling with FPC 3.0+.

RTL with default codepage UTF-8

By default RTL uses the system codepage for AnsiStrings (e.g. FileExists and TStringList.LoadFromFile). On Windows this is a non Unicode encoding, so you can only use characters from your language group (at most 256 characters). On Linux and macOS UTF-8 is typically the system codepage, so the RTL uses here by default CP_UTF8.

FPC since version 3.0 provides an API to change the default system codepage of RTL to something else. Lazarus (actually its LazUtils package) takes advantage of that API and changes it to UTF-8 (CP_UTF8). It means also Windows users now use UTF-8 strings in the RTL.

Usage

Simple rules to follow:

- Normally use type "String" instead of UTF8String or UnicodeString.
- Assign a constant always to a type String variable.
- Use type UnicodeString explicitly for API calls that need it.

These rules make most code already compatible with Delphi when using default project settings.

Using UTF-8 in non-LCL programs

In a non-LCL project add a dependency for **LazUtils** package. Then add **LazUTF8** unit in the uses section of main program file. It must be near the beginning, just after the critical memory managers and threading stuff (e.g. cmem, heaptrc, cthreads).

Calling API functions that use WideString or UnicodeString

When a parameter type is WideString or UnicodeString, you can just pass a String to it. The compiler converts data automatically. There will be a warning about converting from AnsiString to UnicodeString which can be either ignored or suppressed by typecasting the String to UnicodeString.

Variable S in examples below is defined as String which here means AnsiString.

```
procedure ApiCall(aParam: UnicodeString);  // Definition.
 ...
ApiCall(S);                // Call with String S, ignore warning.
ApiCall(UnicodeString(S)); // Call with String S, suppress warning.
```

When a parameter type is a pointer PWideChar, you need a temporary UnicodeString variable. Assign your String to it. The compiler then converts its data. Then typecast the temporary variable to PWideChar.

```
procedure ApiCallP(aParamP: PWideChar);  // Definition.
 ...
var Tmp: UnicodeString;   // Temporary variable.
 ...
```

```
Tmp := S;                  // Assign String -> UnicodeString.
ApiCallP(PWideChar(Tmp)); // Call with temp variable, typecast to pointer.
```

Note, in both cases the code is compatible with Delphi. It means you can copy/paste it to Delphi and it works 100% correctly. In Delphi String maps to UnicodeString.

A typical case is a Windows API call. Only the "W" versions of them should be called because they support Unicode. Normally use UnicodeString also with Windows API. WideString is only needed with COM/OLE programming where the OS takes care of memory management.

Reading / writing text file with Windows codepage

This is not compatible with Delphi nor with former Lazarus code. In practice you must encapsulate the code dealing with system codepage and convert the data to UTF-8 as quickly as possible.

Either use RawByteString and do an explicit conversion

```
uses ... , LConvEncoding;
 ...
var
  StrIn: RawByteString;
  StrOut: String;
 ...
StrOut := CP1252ToUTF8(StrIn,true);  // Use fixed codepage 1252
// or
StrOut := WinCPToUTF8(StrIn,true);  // Use system codepage in this particular computer
```

... or set the right codepage for an existing string

```
var
  StrIn: String;
 ...
SetCodePage(RawByteString(StrIn), 1252, false);  // Codepage 1252 (or
Windows.GetACP())
```

Note: there must be some text in the string variable. An empty string is actually a Nil pointer and you cannot set its codepage.

Windows.GetACP() returns the Windows system codepage.

Code that depends very much on Windows codepage

There is a "plan B" for code that depends very much on Windows system codepage, or must write to Windows console beyond the console codepage.
See: Lazarus with FPC3.0 without UTF-8 mode.
Fortunately it is not needed often. In most cases it is easier to convert data to UTF-8.

Writing to console

Windows console output works **if** your characters belong to the console codepage. For example a box-drawing characters '╩' in codepage CP437 is one byte **#202** and is often used like this:

```
write('╩');
```

When you convert the code to UTF-8, for example by using Lazarus' source editor popup menu item **File Settings / Encoding / UTF-8** and clicking on the dialog button "Change file on disk", the ╩ becomes 3 bytes (#226#149#169), so the literal becomes a *string*. The procedures *write* and *writeln* convert the UTF-8 string to the current console codepage. So your console program now outputs the '╩' on Windows with any codepage (i.e. not only CP437) and it even works on Linux and macOS. You can also use the '╩' in LCL strings, for instance Memo1.Lines.Add('╩');

If your characters do **not** belong to the console codepage, things get trickier. Then you cannot use the new Unicode system at all.
See: Lazarus with FPC3.0 without UTF-8 mode#Problem System encoding and Console encoding .28Windows.29

Unicode characters and codepoints in code

See details for UTF-8 in UTF8 strings and characters.

CodePoint functions for encoding agnostic code

LazUtils package has **unit LazUnicode** with special functions for dealing with codepoints, regardless of encoding. They use the UTF8...() functions from LazUTF8 when used in the UTF-8 mode, and UTF16...() functions from LazUTF16 when used in FPC's {$ModeSwitch UnicodeStrings} or in Delphi (yes, Delphi is supported!).

Currently the {$ModeSwitch UnicodeStrings} can be tested by defining "UseUTF16". There is also a test program LazUnicodeTest in components/lazutils/test directory. It has 2 build modes, UTF8 and UTF16, for easy testing. The test program also supports Delphi, then the UTF-16 mode is used obviously.

LazUnicode allows one source code to work between :

- Lazarus with its UTF-8 solution.
- Future FPC and Lazarus with Delphi compatible UTF-16 solution.
- Delphi, where String = UnicodeString.

It provides these encoding agnostic functions:

- CodePointCopy() - Like UTF8Copy()
- CodePointLength() - Like UTF8Length()
- CodePointPos() - Like UTF8Pos()
- CodePointSize() - Like UTF8CharacterLength() (**Deprecated**. Use UTF8CodepointSize instead. See more)
- UnicodeToWinCP() - Like UTF8ToWinCP()
- WinCPToUnicode() - Like WinCPToUTF8()

It also provides an enumerator for CodePoints which the compiler uses for its for-in loop. As a result, regardless of encoding, this code works:

```
var s, ch: String;
...
for ch in s do
  writeln('ch=',ch);
```

Delphi does not provide similar functions for CodePoints for its UTF-16 solution. Practically most Delphi code treats UTF-16 as a fixed-width encoding which has lead to lots of broken UTF-16 code out there. It means using LazUnicode also for Delphi will improve code quality!

Both units LazUnicode and LazUTF16 are needed for Delphi usage.

String Literals

Sources should be saved in UTF-8 encoding. Lazarus creates such files by default. You can change the encoding of imported files via right click in source editor / File Settings / Encoding.

Usually {$codepage utf8} / -FcUTF8 is not needed. This is rather counter-intuitive because the meaning of that flag is to treat string literals as UTF-8. However the new UTF-8 mode switches the encoding at run-time, yet constants are evaluated at compile-time.

So, without -FcUTF8 the compiler (wrongly) thinks the constant string is encoded with system code page. Then it sees a String variable with default encoding (which will be changed to UTF-8 at run-time but the compiler does not know it). Thus, same default encodings, no conversion needed, the compiler happily copies the characters and everything goes right, while actually it was fooled twice during the process.

Example:

As a rule of thumb, use "String" type and assigning literals works.

```
const s1 = 'äй';
const s2: string = #$C3#$A4; // ä
var s3;
...
  s3 := 'äöü';
```

Note: UTF-8 string can be composed from numbers, as done for s2.

Assigning a string literal to other string types than plain "String" is more tricky. See the tables for what works and what doesn't.

Assign string literals to different string types

Here *working* means correct codepage and correct codepoints. Codepage 0 or codepage 65001 are both correct, they mean UTF-8.

Without {$codepage utf8} or compilerswitch -FcUTF8

String Type, UTF-8 Source	Example	Const (in Source)	Assigned to String	Assigned to UTF8String	Assigned to UnicodeString	Assigned to CP1252String	Assigned to RawByteString	Assigne to ShortStri
const	const s = 'äöü';	working	working	wrong	wrong	wrong	working	working
String	const s: String = 'äöü';	working	working	working	working	working	working	working
ShortString	const s: String[15] = 'äöü';	working	working	working	working	wrong encoded	working	working
UTF8String	const s: UTF8String = 'äöü';	wrong	wrong	wrong	wrong	wrong	wrong	wrong
UnicodeString	const s: UnicodeString = 'äöü';	wrong	wrong	wrong	wrong	wrong	wrong	wrong
String with declared code page	type CP1252String = type AnsiString(1252);	wrong	wrong	wrong	wrong	wrong	wrong	wrong
RawbyteString	const s: RawbyteString = 'äöü';	working	working	working	working	to codepage 0 changed	working	working
PChar	const c: PChar = 'äöü';	working	working	working	working	wrong	working	working

With {$codepage utf8} or compilerswitch -FcUTF8

String Type, UTF-8 Source	Example	Const (in Source)	Assigned to String	Assigned to UTF8String	Assigned to UnicodeString	Assigned to CP1252String	Assigned to RawByteString	Assigne to ShortStri
const	const s = 'äöü';	UTF-16 encoded	working	working	working	working	working	working
String	const s: String = 'äöü';	working	working	working	working	working	working	working
ShortString	const s: String[15] = 'äöü';	wrong	wrong	wrong	wrong	wrong	wrong	wrong
UTF8String	const s: UTF8String = 'äöü';	working	working	working	working	working	working	working
UnicodeString	const s: UnicodeString = 'äöü';	working	working	working	working	working	working	working
String with declared code page	type CP1252String = type AnsiString(1252);	working	working	working	working	working	working	wrong
RawbyteString	const s: RawbyteString = 'äöü';	working	working	working	working	to codepage 0 changed	working	working
PChar	const c: PChar = 'äöü';	wrong	wrong	wrong	wrong	wrong	wrong	wrong

Remember, assignment between variables of different string types always works thanks to their dynamic encoding in FPC 3+. The data is converted automatically when needed.
Only string literals are a challenge.

Coming from older Lazarus + LCL versions

Earlier (before 1.6.0) LCL supported Unicode with dedicated UTF8 functions. The code was not at all compatible with Delphi.

Now many old LCL applications continue to work without changes. However it makes sense to clean the code to make it simpler and more Delphi compatible. Code that reads/writes data with Windows system codepage encoding breaks and must be changed. (See #Reading / writing text file with Windows codepage).

Explicit conversion functions are only needed for I/O with Windows codepage data or when calling Windows Ansi functions. Otherwise FPC takes care of converting encodings automatically. Empty conversion functions are provided to make your old code compile.

- UTF8Decode, UTF8Encode - Almost all can be removed.
- UTF8ToAnsi, AnsiToUTF8 - Almost all can be removed.
- UTF8ToSys, SysToUTF8 - All can be removed. They are now dummy no-ops and only return their parameter.

File functions in RTL now take care of file name encoding. All (?) file name related ...UTF8() functions can be replaced with the Delphi compatible function without UTF8 suffix. For example FileExistsUTF8 can be replaced with FileExists.

Most UTF8...() string functions can be replaced with the Delphi compatible Ansi...() functions. For example UTF8UpperCase() -> AnsiUpperCase().

Now Unicode works in non-GUI programs, too. It only requires a dependency for LazUtils and placing LazUTF8 unit into the uses section of main program file.

For historical reference, this was the old Unicode support in LCL: Old LCL Unicode Support

Technical implementation

What actually happens in the Unicode system? These 2 FPC functions are called in an early initialization section, setting the default String encoding in FPC to UTF-8 :

```
SetMultiByteConversionCodePage(CP_UTF8);
SetMultiByteRTLFileSystemCodePage(CP_UTF8);
```

Under Windows the UTF8...() functions in LazUTF8 (LazUtils) are set as backends for RTL's Ansi...() string functions. Thus those functions work in a Delphi compatible way.

Open issues

- **TFormatSettings** char (bug 27086): for example: **ThousandSeparator**, **DecimalSeparator**, **DateSeparator**, **TimeSeparator**, **ListSeparator**. These should be replaced with string to support UTF-8. For example under Linux with LC_NUMERIC=ru_RU.utf8 the thousand separator is the two byte nbsp/160.
    - **Workaround:** use single space characters instead as done in patch here: 27099

WinAPI function calls in FPC libs

- Unit registry, TRegistry - this unit uses Windows Ansi functions and therefore you need to use UTF8ToWinCP, WinCPToUTF8. Formerly it needed UTF8ToSys.
- All Windows API Ansi function calls in FPC's libraries must be replaced with the "W" function version. This must be done in any case for the future UTF-16 support, thus there is no conflict of interest here.
- **TProcess** - under Windows TProcess FPC 3.0 only supports system codepage. Use either TProcessUTF8 of unit **utf8process** or use FPC trunk where it is fixed, see issue 29136

ToDo: List all related FPC bug tracker issues and patches that should be applied.

Future

The goal of FPC project is to create a Delphi compatible UnicodeString (UTF-16) based solution, but it is not ready yet. It may take some time to be ready.

This UTF-8 solution of LCL in its current form can be considered temporary. In the future, when FPC supports UnicodeString fully in RTL and FCL, Lazarus project will provide a solution for LCL that uses it. At the same time the goal is to preserve UTF-8 support although it may require changes to string types or something. Nobody know the details yet. We will tell when we know...

In essence LCL will probably have 2 versions, one for UTF-8 and one for UTF-16.

FAQ

What about Mode DelphiUnicode?

The **{$mode delphiunicode}** was added in FPC 2.7.1 and is like *{$Mode Delphi}* with *{$ModeSwitch UnicodeStrings}*. See the next question about *ModeSwitch UnicodeStrings*.

What about ModeSwitch UnicodeStrings?

The **{$ModeSwitch UnicodeStrings}** was added in FPC 2.7.1 and defines "String" as "UnicodeString" (UTF-16), "Char" as "WideChar", "PChar" as "PWideChar" and so forth. This affects only the current unit. Other units including those used by this unit have their own "String" definition. Many RTL strings and types (e.g. TStringList) uses 8-bit strings, which require conversions from/to UnicodeString, which are added automatically by the compiler. The LCL uses UTF-8 strings. It is recommended to use UTF-8 sources with or without "-FcUTF8".

Why not use UTF8String in Lazarus?

Short answer: Because the FCL does not use it.

Long answer:

UTF8String is defined in the **system** unit as

```
UTF8String = type AnsiString(CP_UTF8);
```

The compiler always assumes it has UTF-8 encoding (CP_UTF8), which is a multi byte encoding (i.e. 1-4 bytes per codepoint). Note that the [] operator accesses bytes, not characters nor codepoints. Same for UnicodeString, but words instead of bytes. On the other hand a **String** is assumed at compile time to have *DefaultSystemCodePage* (CP_ACP). *DefaultSystemCodePage* is defined at run time, so the compiler conservatively assumes that *String* and *UTF8String* have different encodings. When you assign or combine *String* and *UTF8String* the compiler inserts conversion code. Same for *ShortString* and *UTF8String*.

Lazarus uses the FCL, which uses *String*, so using *UTF8String* would add conversions. If *DefaultSystemCodePage* is not UTF-8 you lose characters. If it is UTF-8 then there is no point to use *UTF8String*.

UTF8String becomes useful when eventually there is an UTF-16 FCL.

Why does UTF8String show strange characters and String works

For example:

```
var s: UTF8String = 'ä';  // with default flags (i.e. no -Fc) this creates garbage
               // even on a UTF-8 Linux system
```

Question: Is it a bug? Answer: No, because it works as documented.

FPC ignores the LANG variable to create on every system the same result. For historical reasons/Delphi compatibility it uses ISO-8859-1 as default. Lazarus prefers UTF-8.

- UTF-8 sources work with *String*, because
  1. FPC does not add conversion code for normal String literals by default.
  2. The *source codepage* is equal to the *runtime codepage*. On Windows LazUTF8 sets it to CP_UTF8.
- UTF8String requires UTF-8 sources. Since FPC 3.0 for UTF8String you must tell the compiler, that the source is UTF8 (*-FcUTF8*, *{$codepage UTF8}*, or save file as UTF-8 with BOM).
  - Note: If you tell the compiler the source encoding UTF-8 it changes all non ASCII string literals of this unit to UTF-16, increasing the size of the binary, adding some overhead and PChar on literals require an explicit conversion. That's why Lazarus does not add it by default.

What happens when I use $codepage utf8?

FPC has very limited UTF-8 support. In fact, FPC only supports storing literals as either "default" encoded 8-bit strings or widestrings. So any non default codepage is converted to widestring, even if it is the system codepage. For example most Linux/Mac/BSD use UTF-8 as system codepage. Passing -Fcutf8 to the compiler will store the string literal as widestring.

At run time the widestring literal is converted. When you assign the literal to an AnsiString the widestring literal is converted using the widestringmanager to the system encoding. The default widestringmanager under Unix simply converts the widechars to chars, destroying any non ASCII character. You must use a widestringmanager like the unit cwstring to get correct conversion. Unit LazUTF8 does that.

See also

- Information about Unicode, codepages, string types and RTL in FPC. Note, that information is not entirely useful with Lazarus UTF-8 system, because from FPC's point of view it is a hack and changes the default codepage.
- FPC Unicode support