Введение

Здесь описывается поддержка Unicode в «программах» Lazarus (консольных или серверных, без графического интерфейса) и «приложениях» (GUI с использованием LCL) при использовании особенностей FPC 3.0+.

Решение является кросс-платформенным и использует кодировку UTF-8, которая отличается от UTF-16 Delphi, но вы можете написать код, полностью совместимый с Delphi на уровне исходного кода, запомнив лишь несколько правил.

Поддержка Unicode включается автоматически для приложений LCL начиная с Lazarus 1.6.0 при компиляции с FPC 3.0+.

Старый метод поддержки UTF-8 в LCL при использовании FPC версий до 2.6.4 включительно, описан здесь: LCL Unicode Support

RTL с кодовой страницей UTF-8 по умолчанию

По умолчанию RTL использует системную кодовую страницу для AnsiStrings (в таких операциях, как например FileExists и TStringList.LoadFromFile). Под Windows это не-Unicode кодировка, поэтому могут использоваться только символы из текущей языковой группы (не более 256 символов). LCL, с другой стороны, работает с кодировкой UTF-8, охватывающей весь диапазон Unicode. Под Linux и macOS UTF-8 обычно является системной кодовой страницей, и здесь RTL использует по умолчанию CP_UTF8.

FPC, начиная с версии 3.0, обеспечивает API для изменения кодовой страницы RTL по умолчанию на другое значение. Lazarus (конкретно пакет LazUtils) использует данный API и изменяет кодовую страницу RTL по умолчанию на UTF-8 (CP_UTF8). Это означает, что пользователи Windows также могут теперь использовать строки UTF-8 в RTL.

- Например, FileExists и StringList.LoadFromFile(Filename) теперь имеют полную поддержку Unicode. См. Полный перечень функций, которые полностью поддерживают Unicode, здесь:

RTL changes

- *AnsiToUTF8, UTF8ToAnsi, SysToUTF8, UTF8ToSys* не работают (не изменяют передаваемых данных). Эти функции обычно использовались для упомянутых выше функций RTL, которые больше не нуждаются в преобразованиях. Относящееся к функциям WinAPI, см ниже.
- Многочисленные вызовы *UTF8Encode* и *UTF8Decode* больше не требуются, потому что такие действия при присваивании значений *UnicodeString* переменным типа String и наоборот компилятор делает автоматически.
- При работе с WinAPI необходимо использовать "W"-функции или пользоваться функциями **UTF8ToWinCP** и **WinCPToUTF8**. То же верно для библиотек, которые до сих пор используют Ansi WinAPI функции. Например, в FPC 3.0 и более ранних версиях в этом нуждается unit registry.
- "String" и "UTF8String" — различные типы. Если вы присваиваете значение *String* переменной типа *UTF8String*, компилятор добавляет код для проверки совпадения кодировок. Это будет стоить дополнительного времени исполнения и увеличит размер кода. Просто используйте *String* вместо *UTF8String*.
- Консоль Windows использует кодировку, которая может отличаться от системной (в случае русского языка это всегда именно так). **writeln** в FPC 3.0+ автоматически преобразует строки UTF-8 в кодовую страницу консоли. Некоторые консольные программы Windows ожидают на входе строки в кодировке консоли и результаты своей работы также выдают в кодовой странице консоли. Для соответствующего преобразования можно использовать функции **UTF8ToConsole** и **ConsoleToUTF8**.

Дополнительная информация о новинках поддержки Unicode в FPC: FPC Unicode support

Использование

Следуйте простым правилам:

- Используйте как обычно тип "String" вместо UTF8String или UnicodeString.
- Всегда присваивайте константу переменной типа String.
- Используйте тип UnicodeString явно для вызовов API, когда это необходимо.

Эти правила делают большую часть кода уже совместимым с Delphi при использовании настроек проекта по умолчанию.

Применение в Lazarus

Новый режим включается автоматически при компиляции FPC 3.0+. Это поведение может быть отключено директивой компиляции **-dDisableUTF8RTL**, детали см. на странице Lazarus with FPC3.0 without UTF-8 mode.

Если вы используете строковые литералы в новом режиме, ваши исходники всегда должны быть в кодировке UTF-8. Однако, ключ **-FcUTF8** на самом деле обычно не требуется. Дополнительные сведения приведены ниже, в разделе "Строковые литералы".

Что же на самом деле происходит в новом режиме? В секции предварительной инициализации вызываются две FPC функции, устанавливающие кодировку строк по умолчанию в исполняющих библиотеках FPC в UTF-8 :

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

Кроме того, функции UTF8...() из LazUTF8 (LazUtils) устанавливаются в качестве функций обратного вызова для функций RTL, имена которых начинаются с Ansi...().

Использование UTF-8 в программах без LCL

В не LCL-проекте добавьте зависимость для пакета **LazUtils**. Затем добавьте модуль **LazUTF8** в секцию uses основного файла программы. Он должен быть в начале, сразу после критических менеджеров памяти и многопоточности (например, cmem, heaptrc, cthreads).

Вызов функций API, которые используют WideString или UnicodeString

Если тип параметра WideString или UnicodeString, вы можете просто передать ему строку. Компилятор преобразует данные автоматически. Появится предупреждение о преобразовании из AnsiString в UnicodeString, которое можно либо проигнорировать, либо подавить, приведя тип String к UnicodeString.

Переменная S в приведенных ниже примерах определяется как String, что здесь означает AnsiString.

```
procedure ApiCall(aParam: UnicodeString);  // Определение
 ...
ApiCall(S);                 // вызов со строкой S, игнорируя предупреждение.
ApiCall(UnicodeString(S)); // вызов со строкой S, подавляя предупреждение (приведение
String к UnicodeString).
```

Когда тип параметра является указателем PWideChar, вам нужна временная переменная UnicodeString. Присвойте ей свою строку. Затем компилятор преобразует эти данные. Затем введите временную переменную в PWideChar.

```
procedure ApiCallP(aParamP: PWideChar);  // Определение
 ...
var Tmp: UnicodeString;   // Временная переменная
 ...
Tmp := S;                 // Присваиваем String -> UnicodeString.
ApiCallP(PWideChar(Tmp)); // Вызов переменной temp, приведение к указателю
```

Примечание: в обоих случаях код совместим с Delphi. Это означает, что вы можете скопировать/вставить его в Delphi, и он сработает на 100% правильно. В Delphi String проецируется в UnicodeString.

Типичным случаем является вызов Windows API. Только должны вызываться их версии "W", потому что они поддерживают Unicode. Обычно используйте UnicodeString также с Windows API. WideString необходим только при программировании COM/OLE, где ОС заботится об управлении памятью.

Чтение / запись текстового файла с кодовой страницей Windows

Это не совместимо ни с Delphi, ни с прежним кодом Lazarus. На практике вы должны инкапсулировать код, связанный с системной кодовой страницей, и преобразовать данные в UTF-8 как можно быстрее.

Либо используйте RawByteString и сделайте явное преобразование...

```
uses ... , LConvEncoding;
 ...
var
  StrIn: RawByteString;
  StrOut: String;
 ...
StrOut := CP1252ToUTF8(StrIn,true);  // Использует фиксированную кодовую страницу
// или
StrOut := WinCPToUTF8(StrIn,true);  // Использует системную кодовую страницу на этом
конкретном компьютере
```

... или установите правильную кодовую страницу для существующей строки

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

Примечание: в строковой переменной должен быть какой-то текст. Пустая строка на самом деле является указателем Nil, и вы не можете установить ее кодовую страницу.

Windows.GetACP() возвращает системную кодовую страницу Windows.

Код, который очень сильно зависит от кодовой страницы Windows

Существует «план B» для кода, который очень сильно зависит от системной кодовой страницы Windows или должен записываться в консоль Windows за пределами кодовой страницы консоли.
См.: Lazarus с FPC3.0 без режима UTF-8.
К счастью, это нужно нечасто. В большинстве случаев проще конвертировать данные в UTF-8.

Запись в консоль

Вывод консоли Windows работает корректно, **если** ваши символы принадлежат кодовой странице консоли. Например, символ для рисования рамки `'╩'` в кодовой странице CP437 составляет один байт **#202** и часто используется так:

```
write('╩');
```

Когда вы конвертируете код в UTF-8, например, используя пункт всплывающего меню редактора исходного кода Lazarus **File Settings (Параметры файла) / Encoding (Кодировка) / UTF-8**, и нажимая кнопку диалога "Change file on disk" (Изменить файл на диске), символ `'╩'` становится 3-х байтным (#226#149#169), поэтому литерал становится *строкой*.

Процедуры *write* и *writeln* преобразуют строку UTF-8 в текущую кодовую страницу консоли. Таким образом, ваша консольная программа теперь выводит `'╩'` в Windows с любой кодовой страницей (т.е. не только с CP437), и она даже работает в Linux и macOS. Вы также можете использовать `'╩'` в строках LCL, например, Memo1.Lines.Add(`'╩'`);

Если ваши символы **не** принадлежат кодовой странице консоли, все усложняется. Тогда вы вообще не сможете использовать новую систему Unicode.
См.: Проблема Системной кодировки и Консольной кодировки (Windows)

Символы Юникода и кодовые точки в коде

См. детали для UTF-8 в UTF8 strings and characters.

Функции кодовых точек для кодирования независимого кода

В пакете LazUtils есть **модуль LazUnicode** со специальными функциями для работы с кодовыми точками, независимо от кодировки. Они используют функции UTF8...() из LazUTF8 при использовании в режиме UTF-8, и функции UTF16...() из LazUTF16 при использовании в {$ModeSwitch UnicodeStrings} FPC или в Delphi (да, Delphi поддерживается!).

В настоящее время {$ModeSwitch UnicodeStrings} можно протестировать, задав "UseUTF16". Также есть тестовая программа LazUnicodeTest в каталоге components/lazutils/test. Она имеет 2 режима сборки, UTF8 и UTF16, для удобства тестирования. Тестовая программа также поддерживает Delphi, тогда, очевидно, используется режим UTF-16.

LazUnicode позволяет одному исходному коду работать между:

- Lazarus с его UTF-8 решением.
- Будущие FPC и Lazarus с Delphi-совместимым решением UTF-16.
- Delphi, где String = UnicodeString.

Это обеспечивается перечисленными ниже функциями, не зависящим от кодирования:

- `CodePointCopy()` - аналогична `UTF8Copy()`
- `CodePointLength()` - аналогична `UTF8Length()`
- `CodePointPos()` - аналогична `UTF8Pos()`
- `CodePointSize()` - аналогична `UTF8CharacterLength()` (функция **устарела**. Вместо нее используйте `UTF8CodepointSize`. См. подробнее)
- `UnicodeToWinCP()` - аналогична `UTF8ToWinCP()`
- `WinCPToUnicode()` - аналогична `WinCPToUTF8()`

Этот режим также предоставляет перечислитель для кодовых точек, который компилятор использует для цикла for-in. В результате, независимо от кодировки, этот код работает:

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

Delphi не предоставляет аналогичные функции для CodePoints в своем решении UTF-16. Практически большая часть кода Delphi рассматривает UTF-16 как кодировку с фиксированной шириной, что привело к возникновению большого количества неработающего кода UTF-16. Это означает, что использование LazUnicode также для Delphi улучшит качество кода!

Для использования Delphi необходимы оба модуля LazUnicode и LazUTF16.

Строковые литералы

Исходный код должен сохраняться в кодировке UTF-8. Lazarus создает такие файлы по умолчанию. Вы можете изменять кодировку импортированных файлов, щелкая правой кнопкой мыши в редакторе исходного кода / File Settings (Настройки файла) / Encoding (Кодировка).

Обычно директива `{$codepage utf8} / -FcUTF8` не требуется. Это довольно нелогично, поскольку значение этого флага заключается в обработке строковых литералов как UTF-8. Однако новый режим UTF-8 переключает кодирование во время выполнения, а константы оцениваются во время компиляции.

Таким образом, без `-FcUTF8` компилятор (ошибочно) считает, что строковая константа кодируется системной кодовой страницей. Затем он видит переменную String с кодировкой по умолчанию (которая будет изменена на UTF-8 во время выполнения, но компилятор этого не знает). Таким образом, то же для кодировки по умолчанию, преобразование не требуется, компилятор с радостью копирует символы, и все идет хорошо, в то время как на самом деле его дважды обманывали в течение процесса.

Пример:

По правилу шаловливых ручек, используйте тип «String» и заставляйте литералы работать.

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

Примечание: Строка UTF-8 может состоять из чисел, как это продемонстрировано для s2.

- Литералы AnsiString/String работают как с директивой {$codepage utf8} / -FcUTF8 , так и без неё.

```
const s: string = 'äй';
```

- Литералы ShortString работоспособны только **без** указания директивы {$codepage utf8} / -FcUTF8. Вы можете сделать следующее:

```
unit unit1;
{$Mode ObjFPC}{$H+}
{$modeswitch systemcodepage} // прекращает действие директивы -FcUTF8
interface
const s: string[15] = 'äй';
end.
```

В качестве альтернативы, возможно использовать строки shortstring с включенной директивой $codepage путем прямого присвоения кодов символов:

```
unit unit1;
{$Mode ObjFPC}{$H+}
{$codepage utf8}
interface
const s: String[15] = #$C3#$A4; // ä
end.
```

- WideString/UnicodeString/UTF8String работают только **с** {$codepage utf8} / -FcUTF8.

```
unit unit1;
{$Mode ObjFPC}{$H+}
{$codepage utf8}
interface
const ws: WideString = 'äй';
end.
```

Присвоение строкового литерала другим строковым типам, кроме простого «String», более мудрено. Смотрите таблицы, что работает, а что нет.

Присвоение строковых литералов различным типам строк

Здесь *работает* означает правильную кодовую страницу и правильные кодовые точки. Кодовая страница 0 или кодовая страница 65001 - обе являются правильными, они означают UTF-8.

Без {$codepage utf8} или переключателя компилятора -FcUTF8

Тип String, исходник в UTF-8	Пример	Const (в исходнике)	Присвоение String	Присвоение UTF8String	Присвоение UnicodeString	Присвоение CP1252String	Присвоение RawByteString	Присво ShortS
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 с объявленной кодовой страницей	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

С {$codepage utf8} или переключателем компилятора -FcUTF8

Тип String, исходник в UTF-8	Пример	Const (в исходнике)	Присвоение String	Присвоение UTF8String	Присвоение UnicodeString	Присвоение CP1252String	Присвоение RawByteString	Присво ShortS
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

Тип String, исходник в UTF-8	Пример	Const (в исходнике)	Присвоение String	Присвоение UTF8String	Присвоение UnicodeString	Присвоение CP1252String	Присвоение RawByteString	Присвс ShortS
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 с объявленной кодовой страницей	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

Помните, что присваивание между переменными разных типов строк всегда работает благодаря их динамическому кодированию в FPC 3+. Данные конвертируются автоматически при необходимости. Только строковые литералы являются проблемой.

Переход из более старых версий Lazarus + LCL

Ранее (до 1.6.0) LCL поддерживал Unicode с выделенными функциями UTF8. Код не был совместим с Delphi.

Сейчас многие старые приложения LCL продолжают работать без изменений. Однако имеет смысл очистить код, чтобы сделать его более простым и более совместимым с Delphi. Код, который читает/записывает данные с использованием кодировки системной кодовой страницы Windows, нарушается и должен быть изменен. (См. Чтение / запись текстового файла с кодовой страницей Windows).

Явные функции преобразования необходимы только для ввода-вывода с данными кодовой страницы Windows или при вызове функций Windows Ansi. В противном случае FPC позаботится о автоматическом преобразовании кодировок. Для преобразования вашего старого кода предусмотрены пустые функции преобразования.

- UTF8Decode, UTF8Encode - Почти все можно удалить.
- UTF8ToAnsi, AnsiToUTF8 - Почти все можно удалить.
- UTF8ToSys, SysToUTF8 - Почти все можно удалить. Теперь они являются пустышками и возвращают только свои параметры.

Файловые функции в RTL теперь заботятся о кодировке имен файлов. Все (?) связанные с именем файла функции ...UTF8() можно заменить на Delphi-совместимую функцию без суффикса UTF8. Например, FileExistsUTF8 можно заменить на FileExists.

Большинство строковых функций UTF8...() можно заменить на совместимые с Delphi функции Ansi...(). Например, UTF8UpperCase() -> AnsiUpperCase().

Теперь Unicode работает и в программах без GUI. Требуется только зависимость от LazUtils и размещение модуля LazUTF8 в разделе uses основного файла программы.

Для исторической справки, это была старая поддержка Unicode в LCL: Old LCL Unicode Support

Техническая реализация

Что на самом деле происходит в системе Unicode? Эти 2 функции FPC вызываются в разделе ранней инициализации, устанавливая кодировку String по умолчанию в FPC в UTF-8:

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

В Windows функции UTF8...() в LazUTF8 (LazUtils) устанавливаются в качестве бэкэндов для строковых функций RTL Ansi...(). Таким образом, эти функции работают совместимым с Delphi способом.

Кодовые страницы FPC

Компилятор (FPC) поддерживает указание кодовой страницы, которое может быть записано через опцию командной строки *-Fc* (т.е. *-Fcutf8*) или эквивалентную директиве *codepage* (т.е. *{$codepage utf8}*). В этом случае, перед тем, как копировать байты, представляющие строковые константы в тексте вашей программы, компилятор будет интерпретировать все символьные данные в соответствии с указанной кодовой страницей. Есть две вещи, которые не стоит упускать из виду:

- На платформах Unix, менеджер широких строк должен быть обязательно включен добавлением юнита *cwstring* в перечень uses. Без него программа не сможет правильно преобразовывать строковые данные во время исполнения.

Менеджер широких строк добавляется по умолчанию в новом режиме UTF-8 RTL, однако это делает программу зависимой от libc и усложняет кросс-компиляцию.

- Компилятор преобразует все строковые константы, содержащие символы, не относящиеся к ASCII, в константы типа widestring. Затем они автоматически преобразуются обратно в ansistring (либо во время компиляции, либо во время исполнения), но это может привести к искажениям, если вы попытаетесь смешать в одной строковой константе символы, напечатанные в тексте исходника и символы, заданные числовым представлением:

Например:

```
program project1;
{$codepage utf8}
{$mode objfpc}{$H+}
{$ifdef unix}
uses cwstring;
{$endif}
var
  a,b,c: string;
begin
  a:='ä';
  b:='='#$C3#$A4; // #$C3#$A4 - закодированный в UTF-8 символ ä
  c:='ä='#$C3#$A4; // после не относящегося к  ascii 'ä' компилятор интерпретирует
#$C3 в качестве отдельного widechar.
  writeln(a,b); // выводит ä=ä
  writeln(c);   // выводит ä=Ã¤
end.
```

После компиляции и исполнения программа выведет:

```
ä=ä
ä=Ã¤
```

Причина в том, что после того как в строке обнаружен символ *ä*, как упоминалось выше, оставшаяся часть строковой константы, присваиваемой переменной 'c', будет обработана как widestring. В результате #$C3 и #$A4 интерпретируются как widechar(#$C3) и widechar(#$A4), вместо прямой трансляции в байтовое представление.

Открытые вопросы

- Символьные переменные в **TFormatSettings** (баг 27086): например: **ThousandSeparator**, **DecimalSeparator**, **DateSeparator**, **TimeSeparator**, **ListSeparator**. Эти поля должны быть заменены на строки для корректной поддержки UTF-8. Например, под Linux с установками LC_NUMERIC=ru_RU.utf8 разделитель тысяч состоит из двух байт nbsp/160.
  - **Обход проблемы:** использовать только одиночные символы пробелов вместо того, что должно быть на самом деле, как это сделано в патче на баг 27099

Вызовы функций WinAPI в библиотеках FPC

- Unit registry, TRegistry - этот unit использует Ansi функции Windows API и поэтому вам придётся пользоваться UTF8ToWinCP, WinCPToUTF8. Раньше для этого требовался вызов UTF8ToSys.
- Все вызовы Windows функций с Ansi API в библиотеках FPC должны быть заменены версией W-API. Это в любом случае будет сделано для будущей поддержки UTF-16, поэтому никакого конфликта интересов нет. (прим. перев. - однако, эти действия означают декларированный отказ от дальнейшей поддержки версий Windows с неполной поддержкой Unicode - Windows 98, 95 и более ранних, а также некоторых мобильных и встроенных)
- **TProcess** - под Windows TProcess FPC 3.0 поддерживает только системную кодовую страницу. Необходимо либо использовать TProcessUTF8 из unit **utf8process** или патчить FPC, см. баг 29136

ToDo: Перечислить все относящиеся к багтрекеру FPC проблемы и патчи, которые могут эти проблемы решить.

Будущее

Целью проекта FPC является создание решения, базирующегося на Delphi-совместимом UnicodeString (UTF-16), но пока мы к этому не готовы. Потребуется длительное время для такой реализации.

Реализацию LCL на базе UTF-8 в её имеющемся виде необходимо рассматривать как временное решение. В будущем, когда в FPC будет полная поддержка UnicodeString как в RTL, так и в FCL, проект Lazarus обеспечит решения для LCL, использующее эти возможности. В то же время целью является и сохранение поддержки UTF-8, несмотря на то, что это может потребовать изменения в строковых типах или чего-то ещё. Деталей пока не знает никто. Мы обязательно сообщим вам о них, когда станет известно...

В сущности, LCL скорее всего придётся в будущем разделиться на две версии - одну для UTF-8, и другую для UTF-16.

ЧаВо

Что насчет режима DelphiUnicode?

**{$Mode delphiunicode}** был добавлен в FPC 2.7.1 и похож на *{$Mode Delphi}* с *{$ModeSwitch UnicodeStrings}*. Смотрите следующий вопрос о *ModeSwitch UnicodeStrings*.

Как насчет ModeSwitch UnicodeStrings?

{$ModeSwitch UnicodeStrings} был добавлен в FPC 2.7.1 и определяет «String» как «UnicodeString» (UTF-16), «Char» как «WideChar», «PChar» как «PWideChar» и так далее. Это влияет только на текущий модуль. Другие модули, в том числе используемые этим модулем, имеют свое собственное определение «String». Многие строки и типы RTL (например, TStringList) используют 8-битные строки, которые требуют преобразования из/в UnicodeString, которые автоматически добавляются компилятором. LCL использует строки UTF-8. Рекомендуется использовать исходники в кодировке UTF-8 с или без "-FcUTF8".

Почему бы не использовать UTF8String в Lazarus?

Кратко: потому что FCL не использует его.

Исчерпывающе: **UTF8String** определен в модуле **system** как

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

Компилятор всегда предполагает, что он имеет кодировку UTF-8 (CP_UTF8), которая является многобайтовой кодировкой (то есть 1-4 байта на кодовую точку). Обратите внимание, что оператор [] обращается к байтам, а не к символам или кодовым точкам. То же самое для UnicodeString, но только слова вместо байтов. С другой стороны, предполагается, что **String** во время компиляции имеет *DefaultSystemCodePage* (CP_ACP). *DefaultSystemCodePage* определяется во время выполнения, поэтому компилятор консервативно полагает, что *String* и *UTF8String* имеют разные кодировки. Когда вы присваиваете или комбинируете *String* и *UTF8String*, компилятор вставляет код преобразования. То же самое для *ShortString* и *UTF8String*.

Lazarus использует FCL, который использует *String*, поэтому использование *UTF8String* добавит конверсии. Если *DefaultSystemCodePage* не UTF-8, вы теряете символы. Если это UTF-8, то нет смысла использовать *UTF8String*.

UTF8String станет полезным, когда в конце концов появится FCL UTF-16.

Почему UTF8String показывает странные символы, а String работает

Например

```
var s: UTF8String = 'ä';// с флагами по умолчанию (т.е. нет -Fc) это создает мусор
                        // даже в системе UTF-8 Linux
```

Вопрос: это ошибка? Ответ: Нет, потому что это работает как задокументировано.

FPC игнорирует переменную LANG для создания в каждой системе одинакового результата. По историческим причинам/Delphi-совместимости он использует ISO-8859-1 по умолчанию. Lazarus предпочитает UTF-8.

- Исходники UTF-8 работают со *String*, потому что
    1. FPC по умолчанию не добавляет код преобразования для обычных строковых литералов.
    2. *Исходная кодовая страница* равна *кодовой странице времени выполнения*. В Windows LazUTF8 устанавливает значение CP_UTF8.
- UTF8String требует исходников UTF-8. Начиная с FPC 3.0 для UTF8String вы должны сообщать компилятору, что исходник является UTF8 (*-FcUTF8*, *{$codepage UTF8}* или сохранять файл как UTF-8 с BOM).

Примечание: Если вы сообщаете компилятору исходную кодировку UTF-8, он заменяет все строковые литералы ASCII этого модуля на UTF-16, увеличивая размер двоичного файла, добавляя некоторые служебные данные, и PChar для литералов требует явного преобразования. Вот почему Lazarus не добавляет его по умолчанию.

Что случится, если я использую {$codepage utf8}?

FPC имеет очень ограниченную поддержку UTF-8. Фактически, FPC поддерживает хранение литералов только как 8-битные строки с кодировкой «по умолчанию» или как widestrings (широкие строки). Поэтому любая кодовая страница не по умолчанию преобразуется в widestrings, даже если это системная кодовая страница. Например, большинство Linux/Mac/BSD используют UTF-8 в качестве системной кодовой страницы. Передача -Fcutf8 в компилятор сохранит строковый литерал как widestrings.

Во время выполнения widestring литерал конвертируется. Когда вы присваиваете литерал AnsiString, widestring литерал преобразуется с помощью widestringmanager в кодировку системы. По умолчанию widestringmanager под Unix просто конвертирует widechars в символы, уничтожая любые символы не ASCII. Вы должны использовать widestringmanager, например, cwstring, чтобы получить правильное преобразование. Модуль LazUTF8 делает это.

См. также

Информация о Unicode, кодовых страницах, строковых типах и RTL в FPC.
Обратите внимание, что эта информация не совсем полезна для системы Lazarus UTF-8, потому что с точки зрения FPC она является хаком и меняет кодовую страницу по умолчанию.
Поддержка Юникода в FPC