Keeping up with Delphi in FPC

Michaël Van Canneyt

December 8, 2023

Abstract

Delphi compatibility has always been important for Free Pascal. However, in certain respects, this compatibility was lacking. Recently, lots of work has been done to improve the Delphi-compatibility of FPC.

1 Introduction

The Free Pascal compiler team has always regarded Delphi compatibility as an important feature of the Free Pascal compiler. This compatibility applies first and foremost to the Pascal language: if Delphi can compile it, then so should Free Pascal. For a long time, Delphi 7 was the version to which Free Pascal was most compatible. But obviously, Delphi evolved: Modern Language features were added such as generics, anonymous functions, attributes and extended RTTI. These features have been developed in FPC - some since a longer time, some recently, but many of these features have not yet been incorporated in an officially released version of Free Pascal. To use them, you must use the development version of FPC.

The Delphi compatibility also applies to the basic units that are distributed with Free Pascal: The basic RTL units of Delphi can also be found in Free Pascal.

Delphi and Free Pascal of course evolve over time, and some of the differences in the RTL units make it difficult to keep code working in both Free Pascal and Delphi:

- Some basic units are not present. The number of units in the Delphi RTL has expanded considerably. Basic functionality is offered in System.IOUtils, System.JSON, System.Threading etc. These units have been added to FPC recently.
- Delphi switched to using dotted (namespaced) units: the prefixes System, Data, Vcl, FMX are used throughout the Delphi codebase since many years.
- Delphi has since many years changed the 'String' alias type so it is an alias for a UnicodeString: a string type in which each element of the string (a character) is a UTF-16 character - it uses 2 bytes.

Needless to say, the Free Pascal team always needs to play catch-up with Delphi - the requirement of compatibility is a one-way street.

This has posed a dilemma for the FPC team: In the first place, FPC wishes to remain backwards compatible with itself. Something which is considered equally (if not more) important than Delphi compatibility. Clearly, switching the string type and starting to use namespaces in the unit names renders the code backwards incompatible.

How to solve this ?

2 The solution

After many years, a solution for this dilemma has been implemented. The idea is simple: use the same codebase to recompile all FPC RTL and Packages code twice. One compile is done with settings which are backwards compatible with FPC itself: no dotted names, string is the 1-byte string. A second compile is done with settings that will generated dotted names, and in which string is the 2-byte string.

This results in 2 sets of units, which cannot be mixed. The user needs to choose which set of units he wishes to use:

- The Free Pascal backwards-compatible set of units,
- The "Recent Delphi"-compatible set of units.

The latter has been dubbed the 'unicode rtl'.

There is a third option, which is to compile everything with a personal set of settings: more about this later.

To put this solution in effect, some compiler work was needed:

- The compiler had to change its view on what constitutes the basic `Char` type – till recently this was hardcoded as a 1-byte character and `AnsiChar` was an alias for `Char`.

  Now, `AnsiChar` is the a basic type, and `Char` is an alias defined in the system unit.

  This allows to let the keyword `String` be either an `AnsiString` or a `UnicodeString` and `Char` will always match the type of a single character in a `String`.

- A compiler 'target' was till recently defined as a combination of the target CPU and the target OS.

  This means that for example the i386-win32 and x86_64-win64 are 2 windows platforms: one 32-bit, one 64 bit. Linux knows even more platforms, just as Darwin (macos) has already 3 platforms.

  This definition of platform needed to be extended, and the notion of 'SubTarget' was introduced in the compiler.

Both changes have been instrumental in making it possible to create the 2 sets of units for the end-user.

3 The SubTarget

The notion of a compiler target was extended: it needs to encapsulate a CPU, an OS and optionally also an arbitrary set of settings.

For the Delphi-compatible unicode rtl this "arbitrary set of settings" would be:

- use UTF16 strings.
- use dotted names.

Defining a subtarget simply means we give a name to this set of settings.

The subtarget is always specified to the compiler with a command-line switch: `-t`.

So if you wished to compile a file for subtarget 'unicodertl', you would specify:

```
fpc -Mdelphi -tunicodetrl myproject.pas
```

The subtarget name is then used in various ways: The first way the subtarget name is used, is in the compiler. When the compiler is compiling for a certain target, it defines a macro `fpctarget`.

The value of this macro is the combination of target CPU and OS, separated by a dash. This macro is used in the configuration file of the compiler:

```
-Fu/usr/local/lib/fpc/$fpcversion/units/$fpctarget
-Fu/usr/local/lib/fpc/$fpcversion/units/$fpctarget/*
-Fu/usr/local/lib/fpc/$fpcversion/units/$fpctarget/rtl
```

So when compiler version 3.1.1 is compiling for i386-linux, the compiler 'sees' the following configuration:

```
-Fu/usr/local/lib/fpc/3.3.1/units/i386-linux
-Fu/usr/local/lib/fpc/3.3.1/units/i386-linuxt/*
-Fu/usr/local/lib/fpc/3.3.1/units/i386-linux/rtl
```

Now, the optional 'Subtarget' is introduced, and one of the effects of using it is that it adds the name of the subtarget to the `fpctarget` macro. That means that when compiler version 3.1.1 is compiling for i386-linux and subtarget '`unicodertl`', the configuration becomes

```
-Fu/usr/local/lib/fpc/3.3.1/units/i386-linux-unicodertl
-Fu/usr/local/lib/fpc/3.3.1/units/i386-linuxt-unicodertl/*
-Fu/usr/local/lib/fpc/3.3.1/units/i386-linux-unicodertl/rtl
```

The search paths of the compiler are suddenly different depending on the subtarget.

As said before, the subtarget is a name of a set of settings. This is made concrete by the second effect of using subtarget, and also explains why it needs to be set on the command-line: When looking for the compiler configuration, the compiler will also look for a configuration file which has the subtarget included in its name.

When compiling with subtarget `unicodertl`

```
fpc -Mdelphi -tunicodetrl myproject.pas
```

The compiler will, in the same places where it looks for `fpc.cfg` also look for a file called

```
fpc-unicodertl.cfg
```

On Unix-like platforms, it will also look for the usual "hidden" file in the user's home dir:

```
.fpc-unicodertl.cfg
```

This second configuration file contains the set of settings that defines the subtarget.

There are 2 things to note about this extra configuration file:

1. It must exist. If it does not exist, the compiler will display an error (however, it can be empty).

2. It is *always* loaded even if the option to not load the default configuration files (`-n`) is specified.

So, how is this used to create a Delphi compatible rtl ? A configuration file with the name `fpc-unicodertl.cfg` is created with the following contents:

```
-dUNICODERTL
-Municodestrings
```

This configuration file is then used to compile the RTL and packages. It defines `UNICODERTL` and it specifies the `unicodestrings` modeswitch, which instructs the compiler that `String` must be interpreted as `UnicodeString`.

Of course, many units contain some low-level code where the size of the `Char` type is very important. The sources have been changed where needed so that the code compiles and functions correctly with both definitions of the `String` and `Char` types.

Not only the sources were changed, but also the compilation and installation process was slightly modified:

We saw earlier that when specifying `-tunicodertl` on target `i386-linux`, the compiler will look for compiled units in the following directories:

```
  -Fu/usr/local/lib/fpc/3.3.1/units/i386-linux-unicodertl
  -Fu/usr/local/lib/fpc/3.3.1/units/i386-linuxt-unicodertl/*
  -Fu/usr/local/lib/fpc/3.3.1/units/i386-linux-unicodertl/rtl
```

That means that the units must also be installed there when installing the compiled units.

The makefiles for the RTL and Packages have been adapted to cater for this. There is now a variable `SUB_TARGET` which configures the Makefiles to call the compiler with the specified subtarget. When installing units, the subtarget name will be appended to the directory name where the units are installed.

4 Using namespaces in filenames

To create units with namespaces in it and the same units but without namespaces, a small trick is used. It should be clear that the Free Pascal team cannot maintain 2 sets of units. The solution put in place uses 1 set of files, but uses an include mechanism to create the second (namespaced) file.

As an example, here is the file for the 'System.Math' unit:

```
 unit System.Math;
 {$DEFINE FPC_DOTTEDUNITS}
 {$i math.pp}
```

The `math.pp` file is changed

```
{$IFNDEF FPC_DOTTEDUNITS}
unit Math;
{$ENDIF FPC_DOTTEDUNITS}

interface
uses
```

```
{$IFDEF FPC_DOTTEDUNITS}
  System.SysUtils;
{$ELSE FPC_DOTTEDUNITS}
  sysutils;
{$ENDIF FPC_DOTTEDUNITS}
```

Note that the uses clause is different.

More changes are of course needed: For example, whenever a fully qualified identifier is used (i.e. an identifier which includes the unit name) the name had to be corrected.

This system allows the FPC team to compile an RTL with namespaced filenames, and a RTL with backwards-compatible filenames.

The above exercise has been done for all units distributed by FPC: For every unit, a namespaced version has been put in place. For units that exist in Delphi, the namespaced filename is identical to the name in Delphi. For units that do not have an equivalent in Delphi, the opportunity has been taken to make the filenames more consistent.

All makefiles and fpmake programs in the Free Pascal distribution have been adapted so they will compile the namespaced units when the `FPC_DOTTEDUNITS=1` option is given on the make command-line.

Till now the policy of the Free Pascal team has been to lowercase the unit filenames on case sensitive filesystems. It meant that you can be sloppy in the uses clause: the case of the unit name did not matter, since the compiler looked for a lowercased version of a file in addition to looking for a file with the same case as used in the uses clause.

This practice has been abandoned for namespaced units: The unit name must now have the correct case on case-sensitive filesystems.

How to find the namespaced version of a unit name ? There is a file that specifies for each non-namespaced unit the namespaced name of the unit. This file can be used to change the unit names. You don't need to do this manually, there is a tool that will do this for you. More about this tool later.

The astute reader will have noticed that generating namespaced units is not done using the newly intruced concept of subtargets. Indeed, the 2 features are orthogonal. The reason is that there is no 'special' subtarget. You create as many subtargets as you want, and for each subtarget you can create a namespaced version of the RTL or a backwards-compatible version.

This means it would be possible to generate a RTL without namespaced units but with `String` equal to `UnicodeString`, or a namespaced RTL with String=AnsiString.

The choice of the FPC team is determined by the requirement of having a FPC-compatible RTL and a Delphi-compatible RTL.

5 Creating the Delphi compatible RTL

When the next major release of Free Pascal is released, the FPC team will create the 2 RTLs mentioned above. But you can already enjoy this increased Delphi compatibility today by compiling the compiler and Delphi-compatible RTL yourself.

How to go about it? The first thing to do is to install git and download the compiler sources. This mechanism has been treated in depth in previous articles.

Assuming git is installed and in your PATH, in a terminal window (on the commandline) you can clone the FPC sources:

```
git clone https://gitlab.com/freepascal.org/fpc/source.git fpc
```

Once done, you can compile and install the latest FPC:

```
cd fpc
make clean all PP=/path/to/FPC/3.2.2
make install PP=/path/to/FPC/3.2.2
```

The PP variable must point to the fpc binary of FPC version 3.2.2 (this is a requirement). Where this fpc binary is installed, depends on your system.

The above will compile and install version 3.3.1 of FPC. You can also use FPCUpdeluxe for this.

Once this is done, you should note where the new version of FPC 3.3.1 was installed.

The next step is to create the configuration file `fpc-unicodertl.cfg` in one of the locations where the compiler looks for the configuration file:

```
-dUNICODERTL
-Municodestrings
```

The easiest strategy is to look for the existing `fpc.cfg` and to create the above file right next to it.

Then, in the terminal with the following make commands, which you must execute in the same location as the previous 'make' commands:

```
make -C rtl clean all PP=/path/to/FPC/3.3.1 SUB_TARGET=unicodertl FPC_DOTTEDUNITS=1
make -C rtl install PP=/path/to/FPC/3.3.1 SUB_TARGET=unicodertl FPC_DOTTEDUNITS=1

make -C packages clean all PP=/path/to/FPC/3.3.1 SUB_TARGET=unicodertl FPC_DOTTEDUNITS=1
make -C packages install PP=/path/to/FPC/3.3.1 SUB_TARGET=unicodertl FPC_DOTTEDUNITS=1
```

Once this is done, you can compile recent Delphi code using the new '-tunicodertl' command-line switch.

6 Converting code to use namespaced units

What if you wish to update your code and make use of the Delphi-compatible RTL ? As shown above, the uses clause of a project must be changed. If you have a lot of units, this means a lot of work.

Luckily, the tools that were used to make the dual-use FPC units are available to you. In the FPC source tree, in directory `utils/dotutils` there is a program called `prefixunits`. You can compile it on the commandline or using lazarus, and use it to convert units to the newly used mechanism.

The following command-line will convert unit `myunit.pas` to `company.myunit.pas` using the same mechanism as shown above:

```
prefixunits -b -k known.txt c:\Temp\myunit.pas -n company
```

A new file varcompany.myunit.pas will be written which includes `myunit.pas`. The uses clause in `myunit.pas` will then be rewritten using the conditional define as shown above. The `-b` option tells the program to make a backup, the `-k` option must be used to point to the file which maps the old unit names to the new names. This file is present in the `dotutils` directory, next to the sources of the `prefixunits` tool.

You can also decide to simply use only the new unit names. Then you specify the '-r' flag (for 'replace'):

```
prefixunits -r -b -k known.txt c:\Temp\myunit.pas -n company
```

In that case, no new file is created, and the uses clause in `myunit.pas` will be replaced with a unit clause that uses only the dotted unit names.

This tool is somewhat rudimentary but does what it is designed to do, and no doubt in time the Lazarus IDE will be extended with a nice GUI tool that performs the same task for all files in your project.

7 What about Pas2js ?

For pas2js, the same namespacing operation has been performed. The `-t` option has also been added to the transpiler command-line options, so the working of the 2 compilers is the same. To improve the Delphi compatibility, support for the Delphi 12 multiline string has been added in addition to the already existing multiline string support in Pas2js.

8 Conclusion

A lot of work has been put into making FPC more delphi compatible: new language features, dotted unit names. The work was largely sponsored by a company that wishes to compile its Delphi program with Free Pascal without having to change the sources of the program. As a result, the Delphi compatibility of Free Pascal has received a boost and all users of Free Pascal can now enjoy an improved Delphi compatibility.

Unlike the various transitions done in Delphi which could break backwards compatibility, The FPC team decides that FPC remains backwards compatible with itself. As a result, users can now choose whether they use these more recent units or not: the choice is always theirs.