Creating a portable Pascal codebase.

Marco van de Voort

May 31, 2009

Current version: 0.02

Versioning

A lot has happened since the original version. Kylix can be considered dead for all practical purposes nowadays. Tiburon will have a gigantic impact on anything that handles string. 2.3.x is a bit more Delphi compatible again, specially in the dispatch COM and olevariant direction. So it was time to update the FAQ, and prepare it for the host of Tiburon related issues that are coming.

v0.01 The original (2005) version without a version number.

v0.02 Updated in 2008

1 Introduction

There is a lot of (Object) Pascal code available on the web, usually for Turbo Pascal and Delphi. Since Free Pascal supports both these dialects, most code should in theory be recompilable with FPC. (and to a lesser degree with other compilers with the appropiate Borland mode) The main showstopper problems are a too big dependancy on 16-bit functionality (TP/BP), or on the Visual part of the VCL (Delphi). A lot of Delphi code is also too Windows dependant for use on other platforms, but this (the Windows dependance) is no real long term problem for FPC/win32. There can be a few small problems though because not all Delphi winapi units are supported at the moment. Currently most code that can be shared between FPC and Delphi are libraries that are not visual, but since Lazarus[1] is maturing fast, this will improve in time.

In early 2000, I started to test FPC with Delphi code, mostly to test the new Delphi compability mode of FPC. Quite soon, it turned out that the Delphi compability of the 1.0.x compiler series was too limited, so I switched to the development series (1.1.x/1.9.x, which will become 2.0.x in the future). Since then I have kept on getting Delphi packages to work with FPC, to test new features, libraries etc. The packages that were regularly tested for FPC compability are:

1. Jedi Winapi headers (current status: 99% working, most of the remain problems are related to dispinterfaces)
2. ICS networking components (current status: win32 works fine with recent release versions of both ICS and FPC, the problems were in the preprocessor area and uses clauses mostly) This package pretty much was the first to get selected, mostly because it was compatible with a lot of Delphi versions. Some work has been done on getting the Kylix port to run on *nix platforms, but this is hard, since a lot of linux specific functionality and artefacts are exist, due to the heavy libc unit dependance)
3. Jedi JCL (large parts working, but this is the most difficult package by far, since it is mostly a collection of all kinds of OS dependant routines)
4. Decal, a STL substitute (compiles with fairly minimal changes, but hard to test without demoes)
5. Jedi SDL headers (Works pretty well)
6. Jedi Math

Other packages were tested on an incidental basis too. Because of all this testing, a set of common problems started to surface which are usually *not* FPC specific. These problems are more about multiplatform and multicompiler design, specially related to preprocessor (defines, ifdef's) symbols, most of the others with adding units to the USES clause without thinking. Most packages don't seem to spend any effort to come up with a decent system behind the preprocessor symbols (DEFINE/IFDEF's).

The most common problems while porting software occur in the following categories:

1. True multiplatform considerations, like processor and OS dependant code.
2. preprocessor symbols.
3. separation of general (Object) Pascal code and visual code. (widgetset dependant[2] is a better term for this category)

... These problems are very common in the Delphi/Pascal world. Most can be avoided relatively easy in the early design stage and hopefully this article increases programmers awareness regarding these issues.

[1] Lazarus (http://lazarus.freepascal.org) is a graphical IDE/RAD for Free Pascal.

[2] A widget is the general term for what is often called "a control" in windows terminology.

2 A generic preprocessor design.

As said in the previous paragraph, one of the primary problems of existing Delphi packages[3] is that there usually is no system with respect to preprocessor defines.

Nearly each unit has some preprocessor defines and compiler switches in the first few lines of the unit. FPC needs a few modifications here to put the FPC compiler in Delphi mode ({$mode Delphi}), if it is to compile delphi code without additional commandline arguments. Usually there are some package specific defines too. The problem is that Delphi packages typically duplicate the few defines they need in all units, often even directly using Delphi VER<xx> defines instead of using a central includefile that defines properties.

The solution is as obvious as simple: a central include file that maps compiler versions to compiler and rtl properties. Preferably two even, one for the libraries, and one (extra) for the demoes/examples with extra directives for final programs. The average reader will probably frown on this trivial detail, but believe me, when porting, this very simple change saves an enormous amount of work. Specially if this is carefully set up, and some standard problems are avoided. The rest of this chapter describes a good way to do this, mostly in the form of do's and don'ts.

This kind of solution is not about FPC compatibility per se. How more complex the Delphi tree becomes (Kylix, Tiburon, .NET, 64-bit), the more worthwhile it is to put some effort into structuring defines.

Note that since the first version of this faq several Jedi projects have adapted to this principle. Examining e.g. the Jedi JCL source is a good way to get an idea how to implement this in practice.

2.1 *Do* use a central includefile with defines in EVERY file

A lot of Delphi programmers have a natural aversion against includefiles. This probably stems from the fact that C uses them heavily, and Borland also doesn't really encourage the usage of includefiles. The gigantically complicated automake and autoconf systems have often scared Delphi programmers into a "anything but that" kind of attitude.

However nearly each project that reaches a certain size starts to use a central includefile when faced with multiple compilers and/or -versions. The contents vary, but usually contain defines that optionally add extra debugging code, turn parts off etc.

The main reason for including the central includefile by default in *every* file, is that when creating a certain package for compiler A, you usually can't predict in which units the includefile is also needed in the future. E.g. when somebody tries to implement compiler/rtl/OS dependant parts for a compiler (version) B, with as little changes as possible. It is simpler to simply give up administrating why unit X needs the includefile, and unit Y doesn't, and simply include it everywhere. This avoids creeping-in of local defines via the backdoor.

2.2 *Don't* make your main sourcecode directly dependant on Delphi version defines

A lot of Delphi packages directly use Delphi version defines in their source code, like

```
{$ifdef ver90}
Dosomething;
{$else}
Dosomethingelse;
{$endif}
```

Some others hide this a bit by using defines like `Delphi4up` or `Delphi4plus` , (the Mike Lischke style that was also used in older Jedi includefiles) but that is the same thing in practice. Conditional code should never interact directly with the Delphi versions. Why? The main reason is that FPC is Delphi 2 in some respect, and Delphi 7 in another respect, but also for Delphi usage this is a bit dangerous; consider the following bit of code:

```
{$ifdef delphi4up}
something=array of integer; // dynamic array
{$else
something =^integer;        // workaround using pointers;
{$endif}
```

What if we encounter a problem with the dynamic array support for this particular construct in say Delphi 5? It is hard to disable dynamic arrays usage for Delphi 5 alone, since the source code assumes D4+ can use dynamic arrays, and *all* defines that interact with `ver<x>` defines and derived `delphi4plus` variants will have to be checked to make exceptions for Delphi 5. Runtime library changes are even more problematic. Sometimes a feature got added with D3, changed to use D4 features like int64 and cardinal, but moved to another unit with D6 or D7, and will be different again with .NET.

In other words: the main problem is that *a certain* `ver<x>`
define stands for a lot of different compiler properties, and the source loses information if you use these defines directly. This problem is illustrated in the following similar source bit:

```
type
{$ifdef Delphi4plus}
  something = array of integer;
```

[3] Some of the above mentioned packages already changed their ways, when they started implementing FPC compability

```
  myfunc    = function (something:integer=50):pointer;
{$else
  something = ^integer;
  myfunc    = function (something:integer):pointer;
{$endif}
..
..
Uses
{$ifdef Delphi4plus}
  variants
{$endif}
,windows;
```

While this example might not be ideal, it does illustrate the basic problem, `Delphi4plus` define is used to trigger both default parameters, variants and dynamic arrays u4sage. Usually the (mis)usage of the `ver<x>` defines are not limited to language features, but also for identifiers exported by system, sysutils and other basic units. It is simply impossible to automatically determine if a certain ifdef is for an identifier, a language feature or something else, without knowing all compilers and compilerversions involved intimately.

The solution is as obvious as simple. Each often used capability of the compiler-library system gets its own preprocessor symbol, and the mapping between `ver<x>` defines and these capabilities is done in the central include file. So everything after Delphi 4 defines HAS_DYNAMIC_ARRAY, HAS_DEFAULT_PARAMS etc. This allows a non Delphi compiler to define e.g. HAS_DEFAULT_PARAMS, but not HAS_DYNAMIC_ARRAY, something that is not possible when `ver<x>` is used directly in source. But it also allows to disable a certain feature for a certain Delphi version (e.g. when it is bugged) without having an effect on the other capabilities of that version, even temporarily for testing purposes only (simply comment `{$DEFINE HAS_DYNAMIC ARRAYS}` in the `{$ifdef ver120}` part of the central includefile)

A typical includefile of a project that uses capabilities instead of Delphi versions might look like this:

```
{$ifdef ver80}
{$define i386}
{$define CPU32}                // 32-bits CPU
{$define ENDIAN_LITTLE}
{$define has_classes}
{$define win16}
{$endif}
..
..
{$ifdef ver140}
{$define has_dynarray}
{$define has_defparam}
{$define i386}
{$define ENDIAN_LITTLE}
{$define CPU32}
{$define win32}
{$define has_stream_permissions}
{$endif}
..
..
{$ifdef FPC}
   {$mode Delphi}   // put in Delphi mode
{$ifndef ver1_0}  { only in   FPC version 2+}
   {$define has_dynarray}
   {$define has_defparam}
   {$define has_interface}
// fpc already defines i386, cpu32/CPU64 and ENDIAN_LITTLE/ENDIAN_BIG for all platforms
{$endif}
{$endif}
```

The earlier bit of source code then becomes:

```
type
{$ifdef HAS_DYNARRAY}
  something = array of integer;
  {$else
  something = ^integer;
{$endif}
{$ifdef HAS_DEFAULT_PARAM}
myfunc    = function (something:integer=50):pointer;
```

```
{$else}
myfunc    = function (something:integer):pointer;
{$endif}
.
..
Uses
{$ifdef HAS_VARIANTS}
  variants
{$endif}
,windows;
```

It is almost too trivial to mention isn't it? However it really saves enormous amounts of time porting packages set up like this to other compilers, custom runtime and classes libraries (VCL substitutes) etc etc. Also keep in mind that porters are often not as familiar with the package as you, and little things that seem trivial to you, might not be for them. It also helps tremendously if e.g. FPC has one feature in the development version, but not yet in the release version.

2.3 *Don't* use {$ifdef Linux} if you mean {$ifdef Unix}

Kylix users often put Kylix specific changes under {$ifdef linux}. Don't fall for this trap, since this often causes both linux kernel specific code and general unix functionality as trivial as backward vs forward slashes under the same {$ifdef linux}, which is far to generic.

Include `{$ifdef Linux}{$define Unix}{$endif}` in your central includefiles, and use Unix if it is generic. Better yet, use `{$ifdef Unix}` even if you don't know for sure if it is linux specific, since there really isn't that much linux specific in the average application source.

2.4 *Don't* assume too much about {$else} cases.

Don't use constructs like:

```
{$ifdef win32}
Do_My_Windows_Thing;
{$else}
Do_My_Linux_Thing;
{$endif}
```

This is even a bigger ***don't*** than the previous one. What to substitute this construct with, depends on how secure you want it to be. As usual, the best way is also the most work:

```
{$undef platformcheck}
{$ifdef Win32}
{$define platformcheck}
Do_My_Window_Thing;
{$endif}
{$ifdef Linux}
{$define platformcheck}
Do_My_Linux_Thing;
{$endif}
{$ifndef Platformcheck}
{$info this code must be implemented for this platform}
{$endif}
```

However you would make most people already very happy if you just used:

```
{$ifdef Win32}
Do_My_Window_Thing;
{$endif}
{$ifdef Unix}
Do_My_Linux_Thing;
{$endif}
```

This way, people porting the source to a new platform will have to quickly check most ifdef linux/unix clauses. But they'll probably have to anyway.

2.5 An alternate approach

Another possible solution is using an external preprocessor. Some packages tried this approach, but the problem is that building the source conveniently means also porting the preprocessor. Also, this forces Lazarus/Delphi users to the commandline. I don't really like this solution, but I can imagine modest use in special cases can have its advantages. Specially concerning uses clauses, which simply tend to become a mess. Also large bodies of EXTERNAL declarations can sometimes become much readable without the `{$ifdef win32}stdcall;{$else}cdecl;{$endif}` appended to every declaration.

The limited FPC macro facility can also be used for these purposes, (by defining the macro's in the central includefile), but this approach has the fundamental problem that it is FPC only.

Combining these two could be a step forward (use FPC macros , but expand them by external preprocessor for non FPC compilers). Since FPC runs on the most targets by far, this means the preprocessor only has to run on linux/i386 and win32.

3 OS and widget set independant design.

90% of making a certain package operating system independant lies in starting out with OS independance in mind, not in incorporting a few changes after the main development is done. While this sounds like a cliche, it is like with many cliche's: it is based on real life and generally there is a lot of truth in it.

OS independance is not really something fundamentally different, but mainly a **state of mind.** Separating OS (in)dependant code, widget dependant and independant code etc. What is commonly considered as OS independance is only the implementation of this state of mind.

Note that Kylix is a particular problem. While Kylix is brilliant from the Borland perspective (keep as much win32 Delphi code running as possible), it often conceals the real truth: Unix is fundamentally different from win32. Kylix is often also too linux specific, which is sane from a Borland perspective (they'll probably never support anything else than Linux), but might not be the best solution to get the most of your codebase. However there are two ways of looking at this:

- "the glass is half empty": Linux is totally different from win32.
- "the glass is half full": If you want to support Linux+win32, then it is only a *really small* extra effort to support most other platforms too. Specially if you start with that attitude from the beginning.

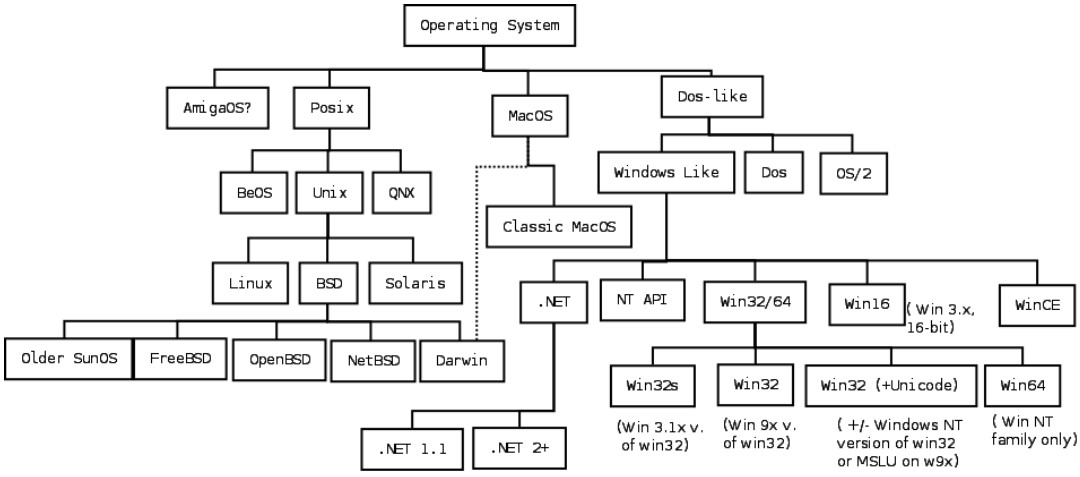
The most important thing is to isolate five basic types of code as much apart as possible.

1. widget set dependant code.
2. OS dependant code
3. Architecture dependant code (processor independant)
4. Generic, in general platform independant code.
5. Compiler implementation dependant code

Specially the second and third category can mix. So there can be some OS dependant code that is also architecture dependant, and some that is not.

Widget set dependant This class of code is the easiest type to define: it is simply code that depends on widget set. In other words, GUI code, code that uses X11, QT, GTK, win32 GDI or Aqua, directly or indirectly. There is a slight gotcha though, code depending on unit FORMS is usually considered GUI code even if it only uses allocatehwnd or the application object, since unit forms is the core of the GUI part of the VCL. Also, some widgetsets (Win32, Aqua) are tied to one platform and one architecture in practice, and some OS dependant entanglement might be hard to avoid in these cases.

OS dependant code OS dependant code simply uses OS dependant features. So unit Windows, ActiveX (the whole WinApi set), Kylix unit Libc etc. However there are gradations when you think of Operating systems in terms of related families. A feature might not be unique for one operating system alone, but for a whole group. The picture below should illustrate this a bit. Specially the difference between Unix and Linux is a mean one, as already discussed.



The picture should be read from top to bottom; POSIX systems have something in common, UNIX systems have a bit more in common, BSDs have even more in common etc. (for the upper levels think about having driveletters or not, forward or backward directory separator, case sensitive/insensitive/preserving filesystem level etc. This is partially what makes Windows and Dos similar) The picture is not really historically correct, but is only meant as the illustration of a possible classification of platforms from a programming perspective. The picture is a gross simplification, and just to give a rough idea, and to show that families ("Unix" or "Windows") are quite complex internally, listing only platforms that are either FPC targets or of general interest. Also the versioning is way more complex than shown above.

One can also imagine GUI widgetset as an added dimension perpendicular to the plane of the above picture. This is not just for Linux (X11,GTK,QT,WxWidgets etc), but also for e.g. OS X (Carbon, COCOA), .NET (Winforms 1.x, 2.x and WPF), Win32/64 (win32 native, MFC).

Architecture dependant code is processor dependant code. Usually assembler, but there are differences between processors that affect the pascal code too:

- endianness (byte ordering, are values stored with the byte with the highest value first, or not)
- 32-bit vs 64-bit architecture (pointers become 64-bit, and might not fit a simple (long) integer anymore, record/structure sizes change)
- alignment requirements (structures/record size change)
- calling convention (not all calling conventions are available for all processors and compilers)
- size of e.g. floating point registers determines which floating point types have decent performance, and how wide they are. Specifically "extended" can be 64,80 or 128 bits. Avoid COMP as the plague, it is too intel specific, and doesn't have much advantages, use int64 as much as possible.

Generic code is code that isn't really dependant on the OS and doesn't heavily depends on architecture. This doesn't mean it doesn't do any I/O at all, but more that all performed I/O is done via OS independant units (think system,sysutils,classes)

Compiler dependant code This relatively small class consists of code that depends on compiler internals. Typical examples are directly accessing the VMT, or assuming too much about the size of a set, calling conventions and exploiting knowledge about the binary format of .EXE files (lowlevel RTTI (introspection), accessing debug metadata and/or sections)

3.1 Directory structure.

Besides conditional compilation, includefiles can also help sorting out the complexity of code with different portability aspects. By simply keeping OS or architecture specific code in a separate directory, selection of the right piece of code is then as easy as making sure the right directories are included while building. This can reduce complexity and amount of ifdefs somewhat. The main problem of this approach is that the way a project is built is not standarized. FPC typically uses GNU Makefiles (though some external projects use scripts/batch files), while Delphi has a build system depending on all kinds of project and configuration files. (which is usually horribly version, installation and OS dependant)

3.1.1 FPC RTL/packages directory structure

An example of how to organize this is the FPC directory structure used in the RTL and packages, that looks like this ($OS is the OS we compile for, $CPU the architecture):

- <pachagename>/$OS (OS dependant stuff goes here)
- <packagename/$OS/$CPU (OS and arch dependanty stuff. Startup code, syscalls, asm interfacing with the OS etc)
- <packagename>/$CPU (CPU specific, but not OS specific, math stuff, strings and language helpers mainly)
- <packagename>/inc (general totally independant include directory. Both .inc as units)
- <packagename>/unix (general shared unix code. Usually 90% of all code for Linux/FreeBSD/ OS X is Unix generic when properly typed and parameterised)
- <packagename>/bsd (if it is a codebase with a lot of OS dependancies, it is even worthwhile to factor out common BSD code, with 4-5 BSD supported)

One can probably easily imagine adding directories for widgetsets (win32, gtk, qt on a basic level, or CLX, VCL, LCL on a higher level) too.

This scheme works very well for libraries that wrap OS dependant functionality in an OS independant way. The avg app can probably do with a subset of these, depending on its nature. (e.g. win32/, unix/, vcl/, clx/ and lcl/). Small differences can better be IFDEFed, but try to do this as general as possible.

4 Project building

As said earlier, the exact build process varies significantly with the compiler, Delphi integrates package/project support in the IDE and uses several files for this, both global (think package registry) and project specific. Worse, these systems are not 100% the same amongst versions.

FPC uses a separate program, GNU Make, and uses a template Makefile.fpc in each build directory that is expanded to the real Makefile by the generator program fpcmake.

At the current time, I'm not aware of efforts to make project building (more) compiler independant. There are two possible approaches:

1. Create a new package information fileformat, and generate both package files from it. (probably D4+ only, since older version have binary package formats)
2. Try to write converters (makefile.fpc to .dpk and vice versa)

Both are not trivial, however can be simplified considerably by moving as much directives as possible to the source. (and of course to the central include file would be best:-) However this is a rough idea, I'd need to know the build system of Delphi much more intimately before pursuing something like this. Waiting till FPC has packages would be wise also, to avoid having to rework the system too soon.

5 Missing/buggy language features in FPC

General problems, or language syntax for which bugs have been reported, but the exact status is not known. These are currently for all known versions:

1. @@xx_is not supported Procvars using double "at" sign are not yet supported in 1.9.2 (I believe now it is supported)
2. The exact state of the "reintroduce" support is not known. (it's ok, it only suppressed a warning)
3. Packages are still missing (see http://wiki.freepascal.org/packages `http://wiki.freepascal.org/packages` for an overview. Shared linking beyond C level in general is poor.
4. In general, FPC is mostly on D7 language level with operator overloading and inline added from later versions.

5.1 FPC 2.0.x

1. **Missing** "implements" functionality (delegation)
2. **Missing** packages
3. **Missing** dispinterfaces and dispid
4. **Missing** custom variants
5. Widestrings and variants need more testing. Widestrings are still missing a lot of RTL routines. Some of the variant wrapper classes are also missed.
6. Currency type used to be on the "buggy" list too. However there has been quite some work on this recently, so it could be ok.

5.2 FPC 2.2.x

1. Dispinterfaces and dispid are mostly working. (or at least parsed correctly)
2. Initial generics support (own syntax, FPC started on this pre Tiburon) that somewhat matured during the 2.2.x cycle. Enough to have e.g. generic container types, but not for advanced generic programming.

5.3 FPC 2.3.x

1. Generics further improved
2. TUnicodeString as a basis for Tiburon unicode string support has been started. It works initially, but is currently internally slightly different from Tiburon, which makes the Tiburon codepage functionality of Ansistring hard to implement. Will probably changed, but development stalled.
3. Dynamic dispatch, customvariants and olevariant improving. Implements delegation still support not entirely complete.
4. Delegation is starting to work but several often used constructs are not implemented yet.

6 Kylix/Libc unit issues

The problem with Kylix is that it is more designed to make recompiling (Win32) Delphi apps for Linux easy than to provide a platform to do generic Unix development. From Borlands perspective, this makes sense, but using Kylix-specific units (mainly unit libc) without wrapper seriously decreases portability.

Unit libc is a direct translation of the libc headers for modern Linux/i386 installations, and thus a lot of unit libc functionality is Linux only, or reveals linux specific undeeded details. There is no attempt at cleaning up the interface presented to the end user. While in theory one could try to emulate certain linux specific calls, the libc-interface was considered too inheritly linux specific and unclean that it was decided to only support unit libc as *legacy* unit on Linux/i386, and modernize the old FPC Unix rtl into something better portable for all Unixy ports. (see the unixrtl.pdf document)

The best way to deal with Kylix libc unit dependancies are:

- Minimize any use of unit libc if possible (e.g. use SysUtils unit and other such units as much as possible).
- It is often better to create a handful of compatible wrapper units for Kylix then the other way around to try to port the Kylix libc units to FPC. So creating a baseunix, termio, dynlibs and users wrappers units for Kylix can often make your source management easier, and keep unnecessary ifdefs from the main source.
- If for some odd ball reason you have to use libc, try to keep the libc unit dependancy localised to as little units as possible. E.g. use a wrapper if you want to use stat64 or so. <functionname>64 functions are a linuxism (and i386 even)
- Please don't ask to port libc to other targets than linux/i386. It won't happen.
- Be very strict about Unix typing, don't assume too much about the sizes of Unix types.
- Do not directly use the dynamic library functions of unit Libc. Always employ a wrapper, preferably modelled or equal to FPC's dynlibs unit (which emerged after problems with Jedi SDL related to dyn loading)
- Do not use the user/group/password functions in unit libc, use the (2.2.2+) package users instead.
- Do not use iconv functions from libc, but use "iconvenc" package (2.2.3+)
- Avoid using messaging systems on *nix. While they may work fine on one type, it is nearly impossible to write generic *nix code with it.
- In general try to avoid exploiting special Linux features to emulate windows behaviour. Make sure the functionality is portable first (e.g. check the Open Group site) , and if you need it despite portability concerns, encapsulate it properly.

7 Misc problems

Comments before {$mode Delphi} must be balanced Only a `{$mode Delphi` puts FPC in Delphi mode. So any comments before this command (or the includefile that contains it) must be balanced, like in TP. This is a **very** common problem.

The exact size that a set occupies can be different FPC currently only supports sets with size 4 and 32 bytes. So everything smaller than 33 elements is 4 bytes in size, 33-256 elements is 32 bytes in size. Also the elements are always counted starting from ordinal value zero. In other words, a `set of 20..40` counts as 41 elements (0..40), even though only 21 (20..40) are used.

Wintypes, Winprocs These old Delphi 1 units can still be used in Delphi. The project file in Delphi aliases these to unit windows. However FPC doesn't have this feature, and doesn't have a project system like this on the compiler level. D1 and win3.x are really too obsolete to create an ugly hack to support this construct. The best way is to always use unit Windows as much as possible, unless Delphi 1 compability is *really* needed (ICS is one of the few packages that still support D1).

Avoid paths in USES clauses Delphi (and FPC) can use syntax like uses xxx IN ' c:\windows\interfaces\sources\include\headers.pas'; This is a plague when porting for obvious reasons, so better avoid it. Relative paths are slightly less evil, but as a whole, packages that use this feature are much more costly to reorganise. Note that since FPC and Delphi have separate build systems above the basic <compiler> <file> level (Delphi with .dpk, FPC using gmake to simplify the buildprocess of large pkgs), the buildprocess always needs some work, and hardcoded paths add an unnecessary complication to deal with.

7.1 Be very picky about what to put in the USES statement

Compared to the good ole Turbo Pascal days, the USES statements in publically available Delphi code are often not cleaned up. Units are USEd that aren't used (:-)), and that has the potential to unnecessary complicate porting. If the unit never used at all, it may even increase your .exe size unnecessarily, because (at least AFAIK) unit initialization/finalization code is always run, and never eliminated by smartlinking. Typically when porting problems of this type arise, the units in questions are winapi units, or VCL units like Graphics. See also the section about Kylix.