# The new Unix RTL.

Marco van de Voort (marco@freepascal.org)

May 26, 2013

## Versions

Current version: 1.4a, just after the 2.6.2 release

**1.0** The version of 2005. No version but the date in the PDF.

**1.1** Unversioned PDF with "June 10th, 2008" as date in it. Mostly adds the "prefix" section.

**1.2** First numbered version.

**1.3** Minor changes, unixtype, libc wiki link

**1.4** more minor changes and updates.

**1.4a** minor fixes to layout, unixutil paragraph

**1.4b** mostly spelling fixes done while committing 1.4a changes.

## 1 Introduction

This is a document where I wrote down some of the reasons for the restructiring of the Unix rtl in the 1.1.x/1.9.x/2.0.x branch[1]. This document was mostly written in retrospect while this branch was maturing, and end-users needed to be prepared for the 1.0->2.0 changes, so it is not really a design document written before the deed.

The restructure was never truly finished, and even now (while preparing for 2.8.0), there are still more things to change, of course the main restructuring has been done, but because some details still have to be done, this document is still relevant I think. The document also tries to explain some of the design considerations behind these changes. Recently, a wiki article http://wiki.freepascal.org/libc_unit was written that shares some of the issues in this doc (e.g. Kylix libc unit issues), and is kept up to date better.

## 2 History

The Unix rtl started life as the Linux rtl. I don't have the exact date, but the design of the Linux unit of the 0.9(9).x and 1.0.x series dates back to 1995-1996, and was made by Michael van Canneyt based on the kernels of that era. (1.1.x, pre-glibc). This rtl was maintained and slightly expanded during the 1996-2000 period, but no fundamental rearrangements were made.

Just before the 1.0 release Marco van de Voort started tinkering with FreeBSD, and the unit linux was a major problem, at least for his skills then :-) However FPC was already too deep into the codefreeze that was needed to stabilize the upcoming 1.0 release to allow a junior member to fully redesign the Unix rtl. FreeBSD had started working when the first betas of 1.0.x were out, and it was mostly a patched Linux RTL. (so it couldn't be committed)

[1]These versions are all the same series. It was called 1.1.x when pre-beta, 1.9.x when in beta stage, and will be 2.0.x when released.

That's why 1.0 was released without formal FreeBSD support, and after cleanup and integration, a 1.0 FreeBSD beta release was delivered a few weeks after the formal release. The minimal modifications for FreeBSD were merged into the CVS system before the 1.0.2 release, and the FreeBSD platform was reasonably established and regarded stable with the release of FPC 1.0.4.

In hindsight however not forcing more fundamental changes at the 1.0.2 release point was a pity. At least a few exotic linux functions should have been banned (sysinfo, clone), and unix typing should have been introduced, and preferably the unit should have been renamed to Unix. Also the syscall interface should have been changed. However besides the conservatism that resulted from the code freeze, I had some doubts about the feasibility of the BSD ports then, and didn't push it hard enough.

During the 1.0.x lifetime, I regretted this deeply, specially after 1.0.6 when other Unix ports appeared, and the FreeBSD port turned out to be qualitatively good. The things that could be solved with a simple IFDEF when the FreeBSD port was done, turned out to be annoying and complicated with multiple ports, and likely to introduce bugs. Small fixes done to the Linux RTL by others constantly broke the BSD ports. The 1.1 branch was made already a year before 1.0. Bug fixes and restructures were only partially ported to the 1.1.x branch, and the OpenBSD/NetBSD/Solaris/QNX/BeOS ports were never ported to 1.1.x.

Because of these facts, there was a lot of maintenance work to do for 1.1.x, and I decided to combine the needed maintenance and updating work with the postponed restructure described above, and along the way tackle as many other problems as possible.

## 3 What's wrong with the situation in 1.0?

Well, there are a lot of reasons actually. Some important ones:

1. The Linux unit was originally targeted at Linux only, 1.x kernels even. Some details "bled" through in the units interface.

2. Quite a lot of different groups of functions with different portability aspects are stuffed together in one unit. This makes porting the complete unit nearly impossible, and also poses some challenges to keep the unit long term compatible on Linux.

3. The Linux unit doesn't have any form of (Unix) typing. Parameters types are translated to Turbo Pascal's integer or longint, using the size in bytes they had in Linux 1.0.x This creates portability problems to other Unices, Linux on other architectures, and makes it harder to fix the unit for newer Linux versions.

4. The error handling of the Linux unit is an own invention, and not compatible with libc, without any major benefits. This complicates the situation when the base library bases on libc.

5. The name is wrong, at least for the current FPC. It doesn't make sense to import an Linux unit under FreeBSD to access Unix functions. Also, in which unit do Linux specific functions end up? "Linux" would be logical, but is already taken.

6. Syscalls that are both used in unit Linux and system were duplicated. Some other units also include these. This adds a small overhead only (typically a few hundreds till several KB) under Linux, but can dramatically increase if wrappers become complex. See also next point.

7. The then-current readdir situation on Linux was bad. Each readdir to get an entry is a syscall, which can be slow. This can sped up by moving parts of the readdir call to userland, and only calling the kernel once in every so and so many blocks (call getdents or getdirentries). Linux implements this too, since the old libc->glibc change, but FPC hadn't caught up yet. The *BSD ports did this from the start, but it is hand coded, and not a translated libc version, which might cause problems with unusual filesystem drivers. (due to AMD64/Linux rtl work, I believe this has been remedied by Peter)

8. (see also 6) The structure of the include files was quite Linux centric, and not very flexible. System and Linux/UNIX unit are too rigidly entangled.

9. Functions weren't named consistently. Some have fd- prefix, some none, some have a slightly different name from libc etc etc. (hmm, this was partially correct in hindsight. fd* functions use the C file type, while the normal (without prefix) are syscalls and use a kernel handle as first argument)

10. (minor) The parameter passing of the syscall interface was system dependent. (Linux: record, BSD: pseudo procedural), this is bad because the syscall interface was exported too.

These reasons are made worse because 2.0 was supposed to support several architectures, and probably more OSes. During the 1.0 lifetime, the Linux unit was ported to *BSD and BeOS, and that already stretched the design to its limits. 2.0 was expected to grow beyond 20 OS-architecture combinations, so that made it much worse. Portability aspects became more important if we wanted to avoid having 2 "good" platforms, and the rest outdated builds that are only partially implemented.

These reasons except number two could be fixed by some major, but doable refactoring of unit Linux, and renaming it to UNIX (as was done originally). However reason two couldn't be fixed this way.

Since full compatibility would be broken by all those other changes anyway, it was decided to do a full redesign, and start from the bottom up, and take care of all these issues, with special attention to ease of maintenance, portability (read: separating portable from unportable code). The design must also scale enough to last for while, 2.0 shipped in 2005, so the 1.0.x series roughly had a lifespan of about 5 years. So a fundamental RTL design must be at least as durable.

### 3.1 Why is it necessary to split up the unit?

The main reasons are related to portability and maintenance. It's easier to do a new port (only the necessary units will be implemented), units will less often be "incomplete" for some targets.

An important side effect is that future source will show more clearly in the USES clause what UNIX functionality is actually used. Use Termio or Syscall is more clear than "Linux". People often think that a single unit is easier than many, but this isn't the case anymore if it stuffed from top till bottom with IFDEFs, and a short description of what function is implemented on what platform is longer than the source code itself.

## 4 What are the basic idea's behind the new 1.1.x/1.9/2.0.x RTL?

1. Introduce Unix typing, so dev_t, off_t etc.

2. Fix the error handling to be compatible with normal Unix (POSIX) errno. *(***Thread safe)**

3. At least keep a possible implementation on top of libc in mind while designing the new RTL. The libraries must be recompilable with a define to keep them syscall free.

4. No more duplication of code. Currently code is duplicated between system and the UNIX/Linux unit.

5. Split up and rename the unit into parts.

   (a) Baseunix which contains the reasonably portable calls (selection loosely based on POSIX)

   (b) Termio which contains the "termio" calls.

   (c) The syscalls moved to the syscall unit.

   (d) The inport, outport calls move to the x86 unit.

   (e) Some very Linux specific calls move to unit Linux. This includes calls like Clone and SysInfo

   (f) Unixutil which contains a few calls that are not Unix specific (usually more general C interfacing). A good place still has to be found for these

   (g) Unix pretty much contains a cleaned up version of the rest.

   (h) If the number of function-categories expands, add additional units instead of adding it to an existing one. E.g. users,sockets,netdb cwstring etc.

6. Functions that have an equivalent in libc are renamed to fp<libcname>. All non fp functions that were added to ease the transition were deprecated in 2.2

7. Introducing a modern readdir will be done too, but as one of the last things to do, since it can be done "under the hood". I believe it was Peter that ultimately did it.

8. Restructuring the includefiles, and detangling the includefiles, and redividing the contents into a platformspecific and -independant parts.

9. The linux syscalls were changed to the BSD way, instead of something that can only be expressed in assembler, the BSDs internally have a pseudo procedural syntax for syscalls. (which is quite generic, probably NetBSD's influence). This spells the end for the syscallreg record that was linux AND x86 centric.

## 4.1 Phasing of the changes.

The restructuring of the code was done in several phases, because the 1.1 branch should remain compilable, so that the compiler developers could keep on working on it. Usually after each phase there was some pauze for stabilising and clean-ups. Roughly these phases were followed:

1. Renaming the linux unit to unix was the first step This sounds trivial, but in practice it turned out to be adding {$ifdef ver1_0} uses linux{$else} uses unix{$endif} for two days. (called *Renamefest* in cvs logs)

2. Restructuring of the syscall interface. This affected both unit Unix and System. All was changed to use the BSD structure as much as possible.

3. At the roughly the same time, the unix typing was introduced.

4. These all needed a lot of cleanup. The BSD ports turned out to be so familiar, that I roughly redivided the BSD rtl between a generic Unix, generic BSD and OS specific part. The BSD rtls share a lot more code now.

5. The system unit was cleaned of linuxisms (mainly sysunix.inc), and parts were made more OS specific

6. A rough first implementation of the baseunix unit was made, using via *external alias* exported syscalls from system. All the rearranging of the includefiles was quite a lot of work. First for FreeBSD, then for Linux.

7. The complete CVS was checked, and changed to use functions from baseunix instead of unit unix. Again (for compiler, fcl, packages, ide) under $IFDEF VER1_0 for bootstrapping reasons. *(Renamefest II* in CVS logs)

8. Functions both in baseunix and unix were removed from unit unix. Unit baseunix was also extended a bit in this phase.

9. Unit unix was cleaned up and split up into multiple units (still in progress)

10. A possibility to recompile unix rtl using libc

11. Cleanup, redividing unix unit over platform (in)dependant includefiles. (mostly done)

12. Darwin port, beos port, more non x86 ports.

## 4.2 Unix errorhandling

The rules of Unix errorhandling are quite easy:

- Each function call indicates somehow if an error occurs. Usually by returning -1. For other functions check the manpages. (typically these functions return a different type then a (C) integer).
- You are only allowed to read the error variable (errno, cerrno, see below) if the function indicates an error.

Besides compability there is another nice thing about this scheme: if an error occurs, one can simply bail out of the function with -1 in some situations, like in the next example:

```
Function somefunc:cint;   // a "unix" function.
Var st : Stat;
Begin
If FpStat('/',st)=-1 Then
  exit(-1);               // exit, errno is already set by fpstat.
... more code...
If FpRmdir('/')=-1 Then
  exit(-1);               // exit, errno is already set by fprmdir.
... more code
  somefunc:=0;
end;
etc etc.
```

This sounds like a shorthand, but there is more to it. If fpstat fills different values on different platforms (or -versions), you simply pass it on.

### 4.2.1 The FPC errorhandling situation, errno and cerrno

FPC normally does its own system calls, and doesn't always to link to libc, which is why the FPC rtl needs an own, independant errorvariable. However when linking to libc or other libraries that use libc it needs access to the libc error variable too. In theory, we could let FPC's syscall write to libc's errno when libc is (also) used, but since that could introduce subtle but hard to trace compability problems, it was decided to keep both errorvariables separate at all times, except when FPC doesn't do syscalls internally at all.

FPC's own errornumber is called errno and is accessable via unit baseunix, libc's errno is accesable via unit initc, and called cerrno. If FPC uses libc for OS interfacing, then both errno's will point to the libc errno.

What does this mean in practice? You need to know if the function you are calling is from a unit that bases on libc calls or *can* also be based on (FPC internal) syscalls. Then select the errorcode (errno,cerrno) accordingly. So if you use unix, linux or similar units, you should get errno (baseunix.fpgeterrno/fpseterrno), if you want to for e.g. unit inet (a typically libc using unit), you need cerrno (initc.fpgetCerrno/fpsetCerrno)

Don't worry about that a syscall using unit uses libc when compiled with FPC_USE_LIBC, that is taken care of properly. (when FPC_USE_LIBC, get/seterrno also update libc's errno)

## 4.3 Libc or syscall?

From time to time, people are asking why FPC isn't using libc, and resorts to syscalls.

There are several reasons for this, but the most important ones were the constant small incompabilities in (Linux) glibc, and the large amount of glibc versions in circulation. (again, mainly for Linux). This includes distributions that compile libc with special options (often legacy free), and then work around this in headers for C users. We even have seen distributions package versions that were officially (according to the glibc site) beta versions.

Moving to use libc by default would mean more than one binary distribution per platform (mainly for Linux, but maybe also for other *nix OSes), without much gain. (the binaries would become slightly larger even when dynlinked with libc, contrary to what you would expect, which is about a 10-40kb. This is probably due to larger libc stubs and relocation data, if PIC is used when linking to libc, the difference might be larger even). Statically linked to libc the binaries are huge. This is mostly because the glibc team probably doesn't prepare for this eventuality.

Not being libc based also avoids some minor binary loader incompabilities that creep up, even if the libc is statically linked.

Another reason is that FPC programs have structures for use with certain functions (like struct STAT) defined in the Pascal rtl, while one calls the C function directly in libc. A C program that calls the same libc function, always uses the right stat because it uses headers supplied with the OS, at least as long as field renaming is consistent[2]. And you'll get a warning if it isn't. But FPC always uses the same one in the RTL. This can be problematic if there are multiple libc's in circulation that use different versions of the structure. (like a 64-bit filesystem version of STAT, and an ordinary one). The kernel doesn't have this, since incompatible versions of the call always get a different syscall number. Sometimes this is possible for libc too (e.g. by always using stat32 or so, or ELF symbol versioning), but the libc situation is generally a bit more difficult. In the stat example, some distributions didn't support stat32 (to force quicker migration to 64-bit fs). And then there are the other unixes to consider.

However all this doesn't meant that an compile option for a libc based rtl isn't nice, since linking to libc can be useful for

- porting purposes (to get the compiler working on a platform for the first time), platforms that are poorly maintained. (QNX, BeOS)
- Darwin (Mac OS X), an OS where the syscalls are said to be a bit more in a state of flux.
- debugging purposes, switch to libc and see if the problem disappears. This works both ways (switch to syscall to detect slight libc incompabilities)
- saving space, e.g programs like Lazarus will link to libc no matter what. Having the RTL link to libc might save a few tens of kb's per application. The exact savings in such case still have to be tested.
- Some functions can be "enhanced" in libc. Specially for security and nameresolving related functionality.

Moreover when done during a large restructure and considered during the design (errno handling), introducing libc support isn't really a lot of work. (initial implementation, generic parts+FreeBSD, about 6-7 hours)

A solution would be a GUI installer (e.g. in Lazarus) that bootstraps FPC, and allows configuring by simply toggling switches. However such an app is a lot of work, and keeps always a bit of a DIY shine. The FreeBSD ports system is also a natural fit, if somebody with enough knowledge of it would step up. But all these require a magnitude more of maintenance.

### 4.3.1 Basic libc Implementation

The units that are primarily affected by libc are system, baseunix and unix. This because these contain a lot of functions that are also in libc, or access these via assembler aliases. Secondary units are nearly all units that are based on syscall, like sockets, ipc etc

A global define FPC_USE_LIBC is introduced that signals "use base functions from libc". (-Ur might be necessary to avoid recompilation). The syscall primitives remain available via unit syscall (and units other than baseunix and unix should use unit syscall and not the aliases)

The 1.0.x compability unit "oldlinux" isn't touched, and always uses syscalls. Since 2.6.0, after 9 years of compatibility existance, it is no longer available precompiled in the default distribution.

[2]And unfortunately automatic unattended conversion of C headers is not really doable.

#### 4.3.2 pipe functions, popen/pclose, a problem?

At first it looked that the pipe functions popen/pclose would become a problem. The FILE type used by these records is the libc internal file structure. Internally these are backed by plain files in libc, and the implementation of these functions in FPC is trivial (using FPC's own internal file record).

A solution proposed by another coremember could be to try keeping the pointer type opague and retrieve the kernel filehandle with fileno() to be able to overload the popen functions with proper pascal filetypes. At least on the platforms where fileno() is a function (and not only a macro). For the closing operation, the FILE pointer should be stored somewhere in the pascal filerecord too.

## 4.4 __errno, __error, _errno_location, h_errno etc.

C is an ancient language which is pretty much frozen due to the enormous amounts of Unix code, and doesn't have an in language threadvar system. However (c)errno is an important global variable that must be threadsafe. This is solved in libc by using some form of macro that usually transforms an errno access to a function call that returns a pointer to the actual errno (right threadinstance) Macro's don't exist after preprocessing, let alone compilation. So when linking to libc we have to poke in the internals, and somehow use the function that returns the pointer to errno directly. This situation is far from ideal, but the problem is made worse by Unix API designers who simply aren't aware of the existance of other languages. (or even C compilers other than the default installed one)

The problem is that the name isn't uniform over platforms, even the free ones. FreeBSD calls it __error, NetBSD __errno and Linux __errno_location. The initc.setcerrno/initc.getcerrno routines wrap this difference.

h_errno is the symbol in the libc library for the non threadsafe variant, and was used in 1.0.x. However this isn't threadsafe, and newer glibc libraries seem to omit it. By default, 1.1.x will use the threadsafe variants, but support for h_errno is still under {$IFDEF }in the initc unit in case you need to work with older libc's.

In general, try to avoid to update C style error variables directly, always use either set/getcerrno or get/seterrno. (the symbols errno and cerrno are ok, these call get/set(c)errno internally)

## 4.5 Exec() functions

The exec() functions have been replaced by the fpexec functions. Moreover, platform independant alternatives like TProcess and ExecuteProcess() are more mature now. The old 1.0.x linux.exec() functions remained in 2.0.x as legacy functions, but were removed starting with 2.2.0

The main idea behind all new functions is the use of "array of ansistring" for the argument of the execl functions. This means a programmer can specify the arguments himself, and are then (with zero copy) passed to the OS. The new way decreases the amount of string operations, and avoids the problems with arguments and filenames that contain spaces that the old functions had. The old functions have been fixed for the most basic quote problems though.

#### 4.5.1 SysUtils.Executeprocess

The new execute functions are used in SysUtils.Executeprocess, which is the new platform independant way of running a program. (comparable to dos.exec, but without the 255 char limit)

Slowly the unit Dos interface is getting increasingly uncomfortable because of shortstrings and dosisms. In general, currently it is recommended to use sysutils as much as possible.

## 4.6 The "FP" prefix

During the past years, I've been pestered about both the need for, and choice of a prefix again and again. The new Unix rtl was written pretty much from scratch, but due to similarity in design requirements resembled Carl's POSIX unit pretty to an high degree. The improved Unix typing was the biggest difference, as well as the splitting up in .inc file that resulted from supporting multiple platforms (posix only supported BeOS afaik).

The prefix was IIRC already in Carl's predecessor the POSIX unit, but there it was "POSIX_". The reason for the prefix was pretty much to have one uniform rule to transform the "C" name to the FPC one. No prefix was dangerous because of clashes with the (then still supported) Linux unit, and long term also for other functions like "exit" "write" and "read" and with OS specific units.

I didn't want anything with "POSIX" in the Unix rtl, because I didn't want to commit outright to POSIX compability due to the possible issues with macroed functionality, and definition on the libc level (vs kernel level). IOW, follow POSIX at armslength, no guarantees it is always an exact 1:1 mapping.

In early versions of the , the prefix was "unx_", but this was considered ugly (and not pascal due to the underscore).

Who and when actually "fp" was decided I don't know. Probably during the BBQ at Rosa and Joerg's place in France, where Carl, Michael and I had a fairly heated discussion about the Unix RTL changes, or the correspondence to work out the details afterwards. It could have been somebody else on IRC even who suggested it. (Florian, Peter or Jonas being the main candidates)

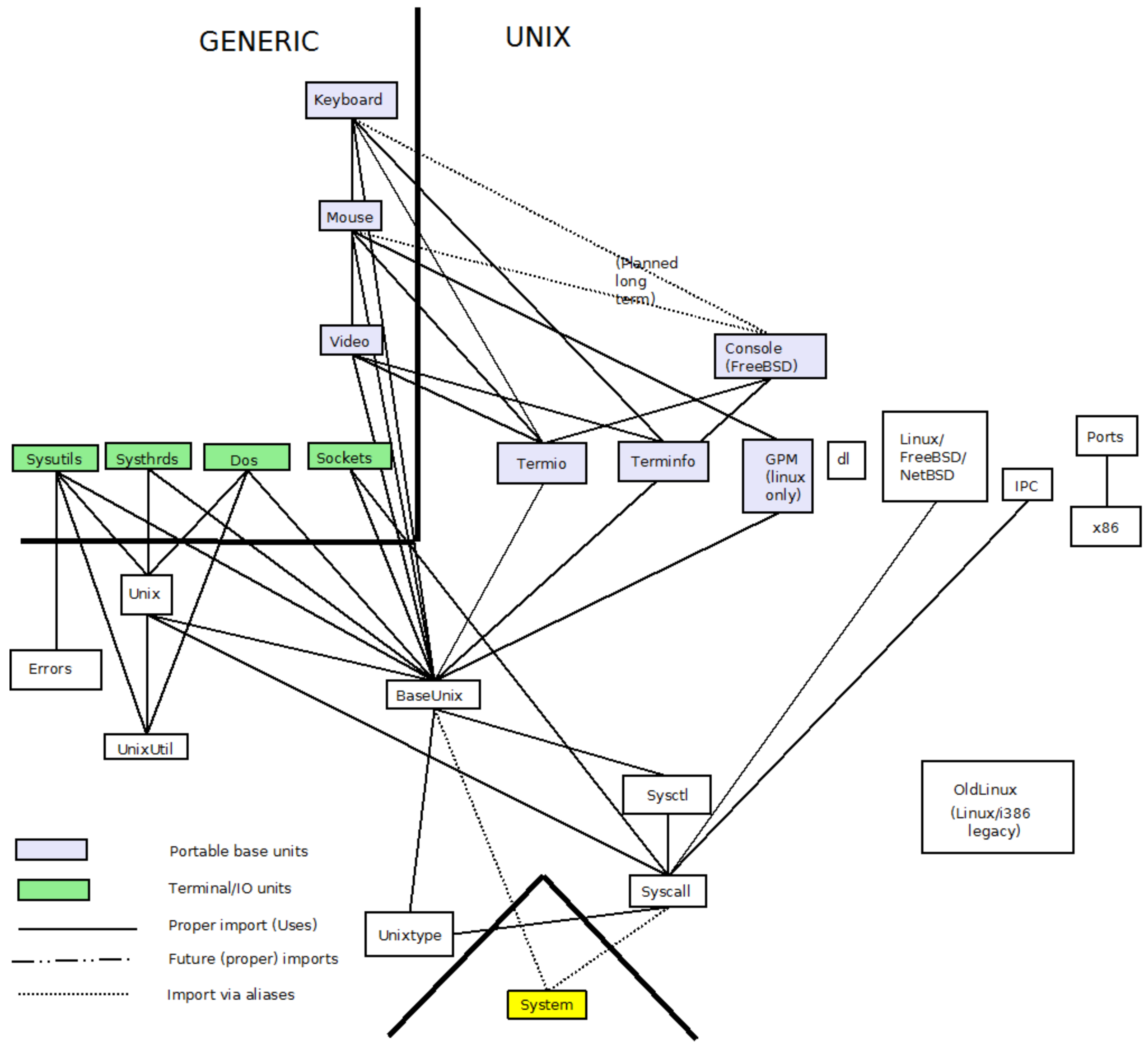
However even in retrospect I still stand by the need to add an prefix, and a short prefix is fine. People often bang on about the "confusion" it will cause, but it would have been much worse IMHO, when one had to explain the non prefixed case: the exceptions, the name clashes with the libc and linux and other platform specific units. Yes, the transition[3] hurt, but IMHO we are in way a better supportable place now.

In my opinion the only way without a prefix would be a Modula-2 like extension that forced all relevant identifiers from the baseunix,unix and socket units to mandatorily prefix with unix name (EXPORT QUALIFIED)

## 5 RTL layout

The following picture tries to explain some of the unit dependancies in the Unix rtl, of course like all documentation, it is probably already outdated :-)

[3]The pains mostly were the "renamefests" and a fat year long dual maintainenance to keep code working with both the 1.0.x and the 1.9.x branch till 2.0 came out. Not only for the FPC project, but also for Lazarus. However with over 5 years between the 1.0 and 2.0 release, there was no way to do it more gradual without compromising 1.0.x internal compatibility. (though IMHO that wouldn't have been a bad thing)



## 5.1 Includefiles

### 5.1.1 Why so many includefiles and ifdefs?

There are many reasons why the FPC rtl is organised as it is. Some of the reasons are:

1. The includefiles allow sharing of code used in multiple places, and that eases maintaining.
2. A higher granularity of the source helps working with CVS'/SVN somewhat. There is less chance that two people work on the same file and have to merge their changes, a problem with units that are thousands of lines.
3. The implementation of a system unit uses a lot of OS dependant types, records and functions that are also used in other units. Includefiles and some tricks allow to reuse the declarations, usually without increasing the size of the binaries. Particularly the Unix system unit exports syscalls via an external alias mechanism. See the separate paragraph about this subject. The main reason for this is to precisely control how many and which symbols the system unit exports, since these are always visible.
4. Exactly what includefiles are OS dependant, and which not, is susceptable to change in the long run. Moving an inc file is easier than totally revising the ifdef system of a huge unit.

5. The system must allow to make exceptions. This is why key units (like System, Baseunix, and in the future unix) are system dependant, but include generic parts. The idea is that a porter can say "I want to implement these parts in a generic way by including the generic includefiles" or "I want to override this functionality with my own code"

6. It allows for the situation where Pascal/Delphi tradition stuffs all related headers in one unit, and still have a file per C header file, which eases header maintenance.

#### 5.1.2 "IMPROPER" exporting from the Unix system unit.

As said in one of the previous paragraphs, the unix system unit exports some OS dependant functions via the [public, alias: 'xxx']; construct. This construct is used to declare names without mangling. This means that some os specific functions in system get a name in a namespace outside the pascal realm, so that other units can import them, like you would from a DLL or from external code. The functions are not exported by normal pascal declarations, thus keeping the interface of the system unit OS-independant. The "client" unit is mainly BaseUnix, but unit Unix also reuses a few functions from system, and exports them.

This is all done to avoid duplication of functions between system on one side and baseunix/unix on the otherside. This saves a few tens of kbs. The types (in ptypes.inc/ctypes.inc) are still imported twice, once in the implementation of system, once in the unixtype unit.

### 5.2 Unixtype

The unit unixtype was introduced pretty late in the rearchitecting proces. Initially baseunix imported ptypes.inc and ctypes, but some platforms needed base unix types below this level (e.g. in header units that were used to implement baseunix). At first these units simply also included ptypes.inc, but this led to type incompability problems once more platforms were implemented[4]. The only solution was to move all types to a separate unit and declare the lowest unit in the RTL (the root of the dependence graph). Since we still wanted to export all unix symbols from baseunix, after some heated discussion, all types in ptypes.inc were aliased. (see aliasptp.inc that aliases ptypes.inc and aliasctp.inc that aliases ctypes.inc)

## 6 Remaining problems

Besides already named problems (e.g. popen), there are some todo's left. Most of these surfaced while porting Kylix apps, and there were some situations where a FPC substitute wasn't easily found:

1. 64-bit file access. The best way to do this, is to simply only have a 64-bit interface, and translate this internally on the few platforms that don't do 64-bit access. Michael deprecated the 32-bit TStreams seek () primitive in 2.6.0.

2. Access to security data (/etc/passwd /etc/groups files etc). Should be extracted and abstract to a separate unit, _with_ a FPC_USE_LIBC option, so that via that avenue users can make sure their apps access via libc, and tie in with all kinds of authentication systems. (there is a header now in the users package since 2.4.2 or so)

3. Improve DNS resolving and accessing. Netdb is not perfect yet, and needs a FPC_USE_LIBC option. (cnetdb added in 2.4.4)

4. Kylixcomp unit for "easy" substitutes for certain constants that ease libc->baseunix porting that we don't want to expose in the proper RTL. (these are hardly used in practice)

5. the unicode primitives are also among the most used functions in unit libc. The widestring manager has some support for these.

6. A lot of the transitional functionality still has to be phased out. See separate paragraph.

[4]The restructure was mostly carried out on FreeBSD, which pretty much only has 32-bit types in the kernel interface. Contrary to linux/x86 that also has 16-bit types

## 6.1 Solved problems

1. unit libc is no longer needed for dynamic loading of libraries (dynlibs)
2. unit libc is no longer needed for basic user/group querying (v2.2.2 “users” package)
3. unit libc is no longer needed for Iconv calls (v2.2.4 iconvenc)
4. the resolver functions of libc are available in unit cnetdb as of 2.4.4
5. A deprecated warning has been added to unit libc in v2.6.2 to avoid design-in in new programs.

## 6.2 Deprecating transitional functionality

A start has been made to remove the helpers and transitional functionality, and a some calls are marked and documented deprecated in 2.2 and 2.2.2 (mantis #0011119), and will be removed in 2.3/2.4. This is a bit hampered by the fact that not all symbols can be marked with deprecated yet.

Some of the deprecated functionality is listed below:

1. 1.0.x fields of the Linux stat record that require a ifdef.
2. non fp socket functions. These were buggy in some cases (formal parameter bug?)
3. Some non fp functions in the unix rtl
4. Unixutil and the functions in it are now in limbo for 3 major versions. See next paragraph

### 6.2.1 unixutil

The remaining routines in this are reusable, and thus should move to something portable. On the other hand this is not possible because non-portable Unix unit depends on them (see RTL dependency graph) A good solution still has to be found for this issue.