Multiplatform Programming Guide

This is a tutorial on writing cross-platform applications with Lazarus and Free Pascal. It will cover the necessary precautions to aid in creating a cross-platform ready program that is ready to deploy.

Most LCL applications work in a cross-platform way without any extra effort. This principle is called „write once, compile anywhere“.

Introduction to Multiplatform (Cross-platform) Programming

How many platforms do you need?

To answer this question, you should first determine who your potential users are and how your program will be used. This question depends on where you are deploying your application.

If you are developing generic desktop software in 2014, Microsoft Windows may be the most important platform. Note that macOS and/or Linux are gaining in popularity, and may be a significant target for your application.

The popularity of the various desktop operating systems differs by country, by the type of software used, and with the target audience; there's no general rule. For example, macOS is quite popular in North America and western Europe, while in South America macOS is mostly restricted to video and sound work.

On many contract projects, only one platform is relevant. Free Pascal and Lazarus are quite capable of writing software targeted at a specific platform. You can, for example, access the full Windows API to write a well integrated Windows program.

If you're developing software that will run on a web server, a Unix platform in one of its various flavors is commonly used. In this case, perhaps only Linux, Solaris, *BSD and other Unixes make sense as your target platforms, although you may want to add support for Windows for completeness.

Once you've addressed any cross-platform issues in your design, you can largely ignore the other platforms, much as you would when developing for a single platform. However, at some point you'll need to test deploying and running your program on the other platforms. For that, it will be helpful to have unrestricted access to machines running the target operating systems. If you don't want multiple physical computers, investigate dual-booting or Virtual Machine (VM) solutions like:

- VMware Fusion for Mac - Host system must be running Intel macOS (Commercial/Free for non-commercial use).
- VMware Workstation Player - Host system running Windows 64 bit or Linux 64 bit (Commercial/Free for personal use).
- VMware Workstation Pro - Host system running Windows 64 bit or Linux 64 bit (Commercial/Free trial).
- Parallels Desktop for Mac - Host system must be running Intel macOS (Commercial/Free trial). Note: There is a Technical Preview that will run on Apple M1 ARM64 processors which will run the ARM versions of Linux and Windows 10 Insider Preview.
- VirtualBox - Host system can be running Intel macOS, Windows, Linux or Solaris (Open Source).

Some, eg Parallels on the Mac, make it trivial to install a desktop Linux operating system with a single click.

See also: Small Virtual Machines and Qemu and other emulators.

Cross-platform Programming

Working with files and folders

When working with files and folders, it is important to use non-platform specific path delimiters and line ending sequences. Here is a list of declared constants in Lazarus to be used when working with files and folders.

- **PathSep**, **PathSeparator**: path separator when adding many paths together (';', ...)
- **PathDelim**, **DirectorySeparator**: directory separator for each platform ('/', '\', ...)
- **LineEnding**: proper line ending character sequence (#13#10 - CRLF, #10 - LF, ...)

Another important thing to be noted is the case sensitiveness of the file system. On Windows filenames are usually not case sensitive, while they usually are case sensitive on Unix platforms (eg Linux, FreeBSD) but not macOS

despite its Unix heritage. But note that if an EXT2, EXT3, etc file system is mounted on Windows, it would be case sensitive. Likewise, a FAT file system mounted on Linux should not be case sensitive.

Special attention should be paid to NTFS which is not case sensitive when used in Windows, but is case sensitive when mounted by POSIX OSes. This could cause **various problems, including loss of files** if files with the same filenames in different cases exist on an NTFS partition mounted in Windows. Using custom functions for checking and preventing creation of several files with the same names on NTFS should be considered by developers.

macOS uses case insensitive filenames by default. This can be the cause of annoying bugs, so any portable application should use filenames consistently.

The RTL file functions use the system encoding for file names. Under Windows this is one of the Windows code pages, while Linux, BSD and macOS usually use UTF-8. The unit **FileUtil** of the LCL provides file functions which takes UTF-8 strings like the rest of the LCL.

```
// AnsiToUTF8 and UTF8ToAnsi need a widestring manager under Linux, BSD, macOS
// but normally these OS use UTF-8 as system encoding so the widestringmanager
// is not needed.
function NeedRTLAnsi: boolean;// true if system encoding is not UTF-8
procedure SetNeedRTLAnsi(NewValue: boolean);
function UTF8ToSys(const s: string): string;     // as UTF8ToAnsi but more independent
of widestringmanager
function SysToUTF8(const s: string): string;     // as AnsiToUTF8 but more independent
of widestringmanager
function UTF8ToConsole(const s: string): string; // converts UTF8 string to console
encoding (used by Write, WriteLn)

// file operations
function FileExistsUTF8(const Filename: string): boolean;
function FileAgeUTF8(const FileName: string): Longint;
function DirectoryExistsUTF8(const Directory: string): Boolean;
function ExpandFileNameUTF8(const FileName: string): string;
function ExpandUNCFileNameUTF8(const FileName: string): string;
function ExtractShortPathNameUTF8(Const FileName : String) : String;
function FindFirstUTF8(const Path: string; Attr: Longint; out Rslt: TSearchRec):
Longint;
function FindNextUTF8(var Rslt: TSearchRec): Longint;
procedure FindCloseUTF8(var F: TSearchrec);
function FileSetDateUTF8(const FileName: String; Age: Longint): Longint;
function FileGetAttrUTF8(const FileName: String): Longint;
function FileSetAttrUTF8(const Filename: String; Attr: longint): Longint;
function DeleteFileUTF8(const FileName: String): Boolean;
function RenameFileUTF8(const OldName, NewName: String): Boolean;
function FileSearchUTF8(const Name, DirList : String): String;
function FileIsReadOnlyUTF8(const FileName: String): Boolean;
function GetCurrentDirUTF8: String;
function SetCurrentDirUTF8(const NewDir: String): Boolean;
function CreateDirUTF8(const NewDir: String): Boolean;
function RemoveDirUTF8(const Dir: String): Boolean;
function ForceDirectoriesUTF8(const Dir: string): Boolean;

// environment
function ParamStrUTF8(Param: Integer): string;
function GetEnvironmentStringUTF8(Index: Integer): string;
function GetEnvironmentVariableUTF8(const EnvVar: string): String;
function GetAppConfigDirUTF8(Global: Boolean): string;

// other
function SysErrorMessageUTF8(ErrorCode: Integer): String;
```

Empty file names and double path delimiters

There are differences in file/directory name handling in Windows versus Linux, Unix and Unix like systems.

- Windows allows empty file names. That's why FileExistsUTF8('..\') checks under Windows in the parent directory for a file without name.

- On Linux/Unix/Unix-like systems, an empty file is mapped to the directory and directories are treated as files. This means that FileExistsUTF8('../') under Unix checks for the existence of the parent directory, which normally results true.

Double path delimiters in file names are also treated differently:

- Windows: 'C:\' is not the same as 'C:\\'
- Unix like OS: the path '/usr//' is the same as '/usr/'. If '/usr' is a directory then even all three are the same.

This is important when concatenating file names. For example:

```
FullFilename:=FilePath+PathDelim+ShortFilename; // can result in two PathDelims which
gives different results under Windows and Linux
FullFilename:=AppendPathDelim(FilePath) + ShortFilename; // creates only one PathDelim
FullFilename:=TrimFilename(FilePath+PathDelim+ShortFilename); // creates only one
PathDelim and do some more clean up
```

The function TrimFilename replaces double path delimiters with single ones and shorten '..' paths. For example /usr//lib/../src is trimmed to /usr/src.

If you want to know if a directory exists use **DirectoryExistsUTF8**.

Another common task is to check if the path part of a file name exists. You can get the path with ExtractFilePath, but this will contain the path delimiter.

- Under Unix like system you can simply use FileExistsUTF8 on the path. For example FileExistsUTF8('/home/user/') will return true if the directory /home/user exists.
- Under Windows you must use the DirectoryExistsUTF8 function, but before that you must delete the path delimiter, for example with the ChompPathDelim function.

Under Unix like systems the root directory is '/' and using the ChompPathDelim function will create an empty string. The function DirPathExists works like the DirectoryExistsUTF8 function, but trims the given path.

Note that Unix/Linux uses the '~' (tilde) symbol to stand for the home directory, typically '/home/jim/' for a user called jim. So '~/myapp/myfile' and '/home/jim/myapp/myfile' are identical on the command line and in scripts. However, the tilde is not automatically expanded by Lazarus. It is necessary to use ExpandFileNameUTF8('~/myapp/myfile') to get the full path.

Text encoding

Text files are often encoded in the current system encoding. Under Windows this is usually one of the windows code pages, while Linux, BSD, and macOS usually use UTF-8. There is no 100% rule to find out which encoding a text file uses. The LCL unit **lconvencoding** has a function to guess the encoding:

```
function GuessEncoding(const s: string): string;
function GetDefaultTextEncoding: string;
```

And it contains functions to convert from one encoding to another:

```
function ConvertEncoding(const s, FromEncoding, ToEncoding: string): string;

function UTF8BOMToUTF8(const s: string): string; // UTF8 with BOM
function ISO_8859_1ToUTF8(const s: string): string; // central europe
function CP1250ToUTF8(const s: string): string; // central europe
function CP1251ToUTF8(const s: string): string; // cyrillic
function CP1252ToUTF8(const s: string): string; // latin 1
...
function UTF8ToUTF8BOM(const s: string): string; // UTF8 with BOM
function UTF8ToISO_8859_1(const s: string): string; // central europe
function UTF8ToCP1250(const s: string): string; // central europe
function UTF8ToCP1251(const s: string): string; // cyrillic
function UTF8ToCP1252(const s: string): string; // latin 1
...
```

For example to load a text file and convert it to UTF-8 you can use:

```
var
  sl: TStringList;
  OriginalText: String;
  TextAsUTF8: String;
```

```
begin
  sl:=TStringList.Create;
  try
    sl.LoadFromFile('sometext.txt'); // beware: this changes line endings to system
line endings
    OriginalText:=sl.Text;

TextAsUTF8:=ConvertEncoding(OriginalText,GuessEncoding(OriginalText),EncodingUTF8);
    ...
  finally
    sl.Free;
  end;
end;
```

And to save a text file in the system encoding you can use:

```
sl.Text:=ConvertEncoding(TextAsUTF8,EncodingUTF8,GetDefaultTextEncoding);
sl.SaveToFile('sometext.txt');
```

Configuration files

You can use the GetAppConfigDir function from SysUtils unit to get a suitable place to store configuration files on different system. The function has one parameter, called Global. If it is True then the directory returned is a global directory, i.e. valid for all users on the system. If the parameter Global is false, then the directory is specific for the user who is executing the program. On systems that do not support multi-user environments, these two directories may be the same.

There is also the GetAppConfigFile which will return an appropriate name for an application configuration file. You can use it like this:

```
ConfigFilePath := GetAppConfigFile(False);
```

Below are examples of the output of default path functions on different systems:

```
program project1;

{$mode objfpc}{$H+}

uses
  SysUtils;

begin
  WriteLn(GetAppConfigDir(True));
  WriteLn(GetAppConfigDir(False));
  WriteLn(GetAppConfigFile(True));
  WriteLn(GetAppConfigFile(False));
end.
```

You can notice that global configuration files are stored on the /etc directory and local configurations are stored in a hidden directory in the user's home directory. Directories whose name begin with a dot (.) are hidden on UNIX and UNIX-like operating systems. You can create a directory in the location returned by GetAppConfigDir and then store configuration files there.

Note: Normal users are not allowed to write to the /etc directory. Only users with administration rights can do this.

Notice that before FPC 2.2.4 the function was using the directory where the application was to store global configurations on Windows.

The output on Windows 98 with FPC 2.2.0:

```
C:\Program Files\PROJECT1
C:\Windows\Local Settings\Application Data\PROJECT1
C:\Program Files\PROJECT1\PROJECT1.cfg
C:\Windows\Local Settings\Application Data\PROJECT1\PROJECT1.cfg
```

The output on Windows XP with FPC 3.0.4:

```
C:\Documents and Settings\All Users\Application Data\project1\
C:\Documents and Settings\user\Local Settings\Application Data\project1\
C:\Documents and Settings\All Users\Application Data\project1\project1.cfg
C:\Documents and Settings\user\Local Settings\Application Data\project1\project1.cfg
```

The output on Windows 7 and Windows 10 and FPC 3.0.4:

```
C:\ProgramData\project1\
C:\Users\user\AppData\Local\project1\
C:\ProgramData\project1\project1.cfg
C:\Users\user\AppData\Local\project1\project1.cfg
```

The output on macOS 10.14.5 with FPC 3.0.4 (violates Apple Guidelines - see below for the correct macOS file locations):

```
/etc/project1/
/Users/user/.config/project1/
/etc/project1.cfg
/Users/user/.config/project1.cfg
```

The output on FreeBSD 12.1 with FPC 3.0.4:

```
/etc/project1/
/home/user/.config/project1/
/etc/project1.cfg
/home/user/.config/project1.cfg
```

The observant will have noticed a difference between the Windows and non-Windows operating systems output - the Windows output for the `WriteLn(GetAppConfigFile(True));` global configuration code includes a subdirectory but the other operating systems do not. To obtain the same results for non-Windows operating systems, you need to include an additional Boolean parameter: `WriteLn(GetAppConfigFile(True,True));`.

Note: The use of UPX interferes with the use of the GetAppConfigDir and GetAppConfigFile functions.

macOS

In most cases configuration files are preference files, which in macOS should be XML files ending with the ".plist" extension and be stored in /Library/Preferences or ~/Library/Preferences with filenames taken from the "Bundle identifier" field in the Info.plist file of the application bundle. Using .config files in the user directory is a violation of the Apple programming guidelines. See the Locating macOS app support, preferences folders article for code which handles this in an Apple-compliant manner.

Data and resource files

A very common question is where to store data files an application might need, such as Images, Music, XML files, database files, help files, etc. Unfortunately there is no cross-platform function to get the best location to look for data files. The solution is to implement differently on each platform using IFDEFs.

Windows

On Windows, application data that the program modifies should not be put in the application's directory (e.g. C:\Program Files\) but in a specific location (see e.g. "Classify Application Data" (*link is broken*). Windows Vista and newer actively enforce this (users only have write access to these directories when using elevation or disabling UAC) but uses a folder redirection mechanism to accommodate older, wrongly-programmed applications. Just reading, not writing, data from application directories would still work but is not recommended.

In short: use such folder:

```
    OpDirLocal:= GetEnvironmentVariableUTF8('appdata')+'\MyAppName';
```

See Windows Programming Tips - Getting Special Folders

Unix/Linux

On most Unixes (like Linux, FreeBSD, OpenBSD, Solaris but **not macOS**), application data files are located in a fixed location, which can be something like: /usr/local/share/app_name or /opt/app_name.

Application data that needs to be written to by the application often gets stored in places like /usr/local/var/app_name, /usr/local/share/app_name or /usr/local/app_name, with appropriate permissions set.

Help files (aka man pages) should be stored in /usr/local/man/man[appropriate manual section number]/app_name>, with appropriate permissions set. Note that some FreeBSD and Linux manual sections differ or do not exist.

User-specific read/write config/data will normally be stored somewhere under the user's home directory (eg in ~/.config/<programname>). Refer to Configuration Files above.

macOS

macOS is an exception among UNIX operating systems. An application is published in a bundle (a directory with ".app" extension) which is treated by the Finder file manager as a file (in a Terminal you can "cd path/myapp.app" or in Finder right click on the app and choose "Show Package Contents"). Your resource files should be located inside the bundle. If the bundle is "path/MyApp.app", then the:

- executable file is "path/MyApp.app/Contents/MacOS/myapp"
- resources directory is "path/MyApp.app/Contents/Resources"

Application supplied data and resource files should generally be stored in the Resources directory of the application bundle. For code which handles this in an Apple compliant manner, see the Locating the macOS application resources directory article.

Application data files generated by the user should be stored in the user's '~/Library/Application Support' directory. For code which handles this in an Apple compliant manner, see the Locating macOS app support, preferences folders article.

Warning: Never use `paramStr(0)`, or any function which uses it, on any UNIX platform to *determine* the location of the executable, as this is a DOS-Windows-OS/2 convention and has several conceptual problems, which cannot be solved using emulation on other platforms. The only thing `paramStr(0)` is guaranteed to return on UNIX platforms is the name with which the program was started. The directory in which it is located and the name of the actual binary (in case it was started using a symbolic link) are not guaranteed to be available via `paramStr(0)`. For macOS, you can **reliably** locate the application bundle directory using the native code in this article.

32/64 bit

Detecting bitness at runtime

While you can control whether you compile for 32 or 64 bit with compiler defines, sometimes you want to know what bitness the operating system runs. For example, if you are running a 32 bit Lazarus program on 64 bit Windows, you might want to run an external program in a 32 bit program files directory, or you might want to give different information to users: I need this in my LazUpdater Lazarus installer to offer the user a choice of 32 and 64 bit compilers. Code: Detect Windows x32-x64 example.

Detecting bitness of external library before loading it

When you want to load functions from dynamic library into your program, it has to have same bitness as your application. On 64 bit Windows, your application might be 32-bit or 64-bit, and there can be 32-bit and 64-bit libraries on your system. So you might want to check whether dll's bitness is same as your application's bitness before loading the dll dynamically. Here is a function which tests dll's bitness (contributed in forum by GetMem):

```
uses {..., } JwaWindows;

function GetPEType(const APath: WideString): Byte;
const
  PE_UNKNOWN = 0; //if the file is not a valid dll, 0 is returned
 // PE_16BIT   = 1; // not supported by this function
  PE_32BIT   = 2;
  PE_64BIT   = 3;
var
  hFile, hFileMap: THandle;
  PMapView: Pointer;
  PIDH: PImageDosHeader;
  PINTH: PImageNtHeaders;
  Base: Pointer;
```

```
begin
  Result := PE_UNKNOWN;

  hFile := CreateFileW(PWideChar(APath), GENERIC_READ, FILE_SHARE_READ, nil,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
  if hFile = INVALID_HANDLE_VALUE then
  begin
    CloseHandle(hFile);
    Exit;
  end;

  hFileMap  := CreateFileMapping(hFile, nil, PAGE_READONLY, 0, 0, nil);
  if hFileMap = 0 then
  begin
    CloseHandle(hFile);
    CloseHandle(hFileMap);
    Exit;
  end;

  PMapView := MapViewOfFile(hFileMap, FILE_MAP_READ, 0, 0, 0);
  if PMapView = nil then
  begin
    CloseHandle(hFile);
    CloseHandle(hFileMap);
    Exit;
  end;

  PIDH := PImageDosHeader(PMapView);
  if PIDH^.e_magic <> IMAGE_DOS_SIGNATURE then
  begin
    CloseHandle(hFile);
    CloseHandle(hFileMap);
    UnmapViewOfFile(PMapView);
    Exit;
  end;

  Base := PIDH;
  PINTH := PIMAGENTHEADERS(Base + LongWord(PIDH^.e_lfanew));
  if PINTH^.Signature = IMAGE_NT_SIGNATURE then
  begin
    case PINTH^.OptionalHeader.Magic of
      $10b: Result := PE_32BIT;
      $20b: Result := PE_64BIT
    end;
  end;

  CloseHandle(hFile);
  CloseHandle(hFileMap);
  UnmapViewOfFile(PMapView);
end;

//Now, if you compile your application for 32-bit and 64-bit windows, you can check if
dll's bitness is same as your application's:
function IsCorrectBitness(const APath: WideString): Boolean;
begin
  {$ifdef CPU32}
    Result := GetPEType(APath) = 2; //the application is compiled as 32-bit, we ask if
GetPeType returns 2
  {$endif}
  {$ifdef CPU64}
    Result := GetPEType(APath) = 3; //the application is compiled as 64-bit, we ask if
GetPeType returns 3
  {$endif}
end;
```

Pointer / Integer Typecasts

Pointers under 64bit need 8 bytes instead of 4 on 32bit. The 'Integer' type remains 32bit on all platforms for compatibility. This means you can not typecast pointers into integers and back.

FPC defines two types for this: PtrInt and PtrUInt. PtrInt is a 32bit signed integer on 32 bit platforms and a 64bit signed integer on 64bit platforms. The same for PtrUInt, but *unsigned* integer instead.

Use for code that should work with Delphi and FPC:

```
{$IFNDEF FPC}
type
  PtrInt = integer;
  PtrUInt = cardinal;
{$ENDIF}
```

Replace all **integer(SomePointerOrObject)** with **PtrInt(SomePointerOrObject)**.

Endianess

Intel platforms are little endian, that means the least significant byte comes first. For example the two bytes of a word $1234 is stored as $34 $12 on little endian systems. On big endian systems like the powerpc the two bytes of a word $1234 are stored as $12 $34. The difference is important when reading files created on other systems.

Use for code that should work on both:

```
{$IFDEF ENDIAN_BIG}
...
{$ELSE}
...
{$ENDIF}
```

The opposite is ENDIAN_LITTLE.

The system unit provides plenty of endian converting functions, like SwapEndian, BEtoN (big endian to current endian), LEtoN (little endian to current endian), NtoBE (current endian to big endian) and NtoLE (current endian to little endian).

Libc and other special units

Avoid legacy units like "oldlinux" and "libc" that are not supported outside of linux/i386.

Assembler

Avoid assembler.

Compiler defines

```
{$ifdef CPU32}
...write here code for 32 bit processors
{$ENDIF}
{$ifdef CPU64}
...write here code for 64 bit processors
{$ENDIF}
```

Projects, packages and search paths

Lazarus projects and packages are designed for multi platforms. Normally you can simply copy the project and the required packages to another machine and compile them there. You don't need to create one project per platform.

Some advice to achieve this

The compiler creates for every unit a ppu with the same name. This ppu can be used by other projects and packages. The unit source files (e.g. unit1.pas) should not be shared. Simply give the compiler a unit output directory where to create the ppu files. The IDE does that by default, so nothing to do for you here.

Every unit file must be part of **one** project or package. If a unit file is only used by a single project, add it to this project. Otherwise add it to a package. If you have not yet created a package for your shared units, see here:

Creating a package for your common units

Every project and every package should have **disjunct directories** - they should not share directories. Otherwise you must be an expert in the art of compiler search paths. If you are not an expert or if others who may use your project/package are not experts: do not share directories between projects/packages.

Platform specific units

For example the unit wintricks.pas should only be used under Windows. In the uses section use:

```
uses
  Classes, SysUtils
  {$IFDEF Windows}
  ,WinTricks
  {$ENDIF}
  ;
```

If the unit is part of a package, you must also select the unit in the package editor of the package and disable the *Use unit* checkbox.

See also Platform specific units

Platform specific search paths

When you target several platforms and access the operating system directly, then you will quickly get tired of endless IFDEF constructions. One solution that is used often in the FPC and Lazarus sources is to use include files. Create one sub directory per target. For example win32, linux, bsd, darwin. Put into each directory an include file with the same name. Then use a macro in the include path. The unit can use a normal include directive.

An example for one include file for each LCL widget set:

Create one file for each widget set you want to support:

```
win32/example.inc
gtk/example.inc
gtk2/example.inc
carbon/example.inc
```

You do not need to add the files to the package or project. Add the include search path *$(LCLWidgetType)* to the compiler options of your package or project.

In your unit use the directive: {$I example.inc}

Here are some useful macros and common values:

- LCLWidgetType: win32, gtk, gtk2, qt, carbon, fpgui, nogui
- TargetOS: linux, win32, win64, wince, freebsd, netbsd, openbsd, darwin (many more)
- TargetCPU: i386, x86_64, arm, powerpc, sparc
- SrcOS: win, unix

You can use the $Env() macro to use environment variables.

And of course you can use combinations. For example the LCL uses:

```
$(LazarusDir)/lcl/units/$(TargetCPU)-$(TargetOS);$(LazarusDir)/lcl/units/$(TargetCPU)-$(TargetOS)/$(L
```

See here the complete list of macros: IDE Macros in paths and filenames

Machine / User specific search paths

For example you have two windows machines stan and oliver. On stan your units are in *C:\units* and on oliver your units are in *D:\path*. The units belong to the package *SharedStuff* which is *C:\units\sharedstuff.lpk* on stan and *D:\path\sharedstuff.lpk* on oliver. Once you opened the lpk in the IDE or by lazbuild, the path is automatically stored in its configuration files (packagefiles.xml). When compiling a project that requires the package *SharedStuff*, the IDE and lazbuild knows where it is. So no configuration is needed.

If you have want to deploy a package over many machine or for all users of a machine (e.g. a pool for students), then you can add a lpl file in the lazarus source directory. See packager/globallinks for examples.

Locale differences

Some functions from Free Pascal, like StrToFloat behave differently depending on the current locale. For example, in the USA the decimal separator is usually ".", but in many European and South American countries it is ",". This can be a problem as sometimes it is desired to have these functions behave in a fixed way, independently from the locale. An example is a file format with decimal points that always needs to be interpreted the same way.

The next sections explain how to do that.

macOS

Refer to the Locale settings for macOS article for details of setting the locale on macOS.

StrToFloat

A new set of format settings which set a fixed decimal separator can be created with the following code:

```
// in your .lpr project file
uses
...
{$IFDEF UNIX}
clocale
{ required on Linux/Unix for formatsettings support. Should be one of the first
(probably after cthreads?}
{$ENDIF}
```

and:

```
// in your code:
var
  FPointSeparator, FCommaSeparator: TFormatSettings;
begin
  // Format settings to convert a string to a float
  FPointSeparator := DefaultFormatSettings;
  FPointSeparator.DecimalSeparator := '.';
  FPointSeparator.ThousandSeparator := '#';// disable the thousand separator
  FCommaSeparator := DefaultFormatSettings;
  FCommaSeparator.DecimalSeparator := ',';
  FCommaSeparator.ThousandSeparator := '#';// disable the thousand separator
```

Later on you can use this format settings when calling StrToFloat, like this:

```
// This function works like StrToFloat, but simply tries two possible decimal
separator
// This will avoid an exception when the string format doesn't match the locale
function AnSemantico.StringToFloat(AStr: string): Double;
begin
  if Pos('.', AStr) > 0 then Result := StrToFloat(AStr, FPointSeparator)
  else Result := StrToFloat(AStr, FCommaSeparator);
end;
```

Gtk2 and masking FPU exceptions

Gtk2 library changes the default value of FPU (floating point unit) exception mask. The consequence of this is that some floating point exceptions do not get raised if Gtk2 library is used by the application. That means that, if for example you develop a LCL application on Windows with win32/64 widgetset (which is Windows default) and plan to compile for Linux (where Gtk2 is default widgetset), you should keep this incompatibilities in mind.

After this forum topic and answers on this bug report it became clear that nothing can be done about this, so we must know what actually these differences are.

Therefore, let's do a test:

```
uses
  ..., math,...

{...}
```

```
var
  FPUException: TFPUException;
  FPUExceptionMask: TFPUExceptionMask;
begin
  FPUExceptionMask := GetExceptionMask;
  for FPUException := Low(TFPUException) to High(TFPUException) do begin
    write(FPUException, ' - ');
    if not (FPUException in FPUExceptionMask) then
      write('not ');

    writeln('masked!');
  end;
  readln;
end.
```

Our simple program will get what FPC default is:

```
 exInvalidOp - not masked!
 exDenormalized - masked!
 exZeroDivide - not masked!
 exOverflow - not masked!
 exUnderflow - masked!
 exPrecision - masked!
```

However, with Gtk2, only exOverflow is not masked.

The consequence is that EInvalidOp and EZeroDivide exceptions do not get raised if the application links to Gtk2 library! Normally, dividing non-zero value by zero raises EZeroDivide exception and dividing zero by zero raises EInvalidOp. For example the code like this:

```
var
  X, A, B: Double;
// ...

try
  X := A / B;
  // code block 1
except
  // code block 2
end;
// ...
```

will take different direction when compiled in application with Gtk2 widgetset. On win widgetset, when B equals zero, an exception will get raised (EZeroDivide or EInvalidOp, depending on whether A is zero) and "code block 2" will be executed. On Gtk2 X becomes Infinity, NegInfinity, or NaN and "code block 1" will be executed.

We can think of different ways to overcome this inconsistency. Most of the time you can simply test if B equals zero and don't try the dividing in that case. However, sometimes you will need some different approach. So, take a look at the following examples:

```
uses
  ..., math,...

//...
var
  X, A, B: Double;
  Ind: Boolean;
// ...
try
  X := A / B;
  Ind := IsInfinite(X) or IsNan(X); // with gtk2, we fall here
except
  Ind := True; // in windows, we fall here when B equals zero
end;
if Ind then begin
  // code block 2
end else begin
```

```
  // code block 1
end;
// ...
```

Or:

```
uses
  ..., math,...

//...
var
  X, A, B: Double;
  FPUExceptionMask: TFPUExceptionMask;
// ...

try
  FPUExceptionMask := GetExceptionMask;
  SetExceptionMask(FPUExceptionMask - [exInvalidOp, exZeroDivide]); // unmask
  try
    X := A / B;
  finally
    SetExceptionMask(FPUExceptionMask); // return previous masking immediately, we
must not let Gtk2 internals to be called without the mask
  end;
  // code block 1
except
  // code block 2
end;
// ...
```

Be cautious, do not do something like this (call LCL with still removed mask):

```
try
  FPUExceptionMask := GetExceptionMask;
  SetExceptionMask(FPUExceptionMask - [exInvalidOp, exZeroDivide]);
  try
    Edit1.Text := FloatToStr(A / B); // NO! Setting Edit's text goes down to widgetset
internals and Gtk2 API must not be called without the mask!
  finally
    SetExceptionMask(FPUExceptionMask);
  end;
  // code block 1
except
  // code block 2
end;
// ...
```

But use an auxiliary variable:

```
try
  FPUExceptionMask := GetExceptionMask;
  SetExceptionMask(FPUExceptionMask - [exInvalidOp, exZeroDivide]);
  try
    X := A / B; // First, we set auxiliary variable X
  finally
    SetExceptionMask(FPUExceptionMask);
  end;
  Edit1.Text := FloatToStr(X); // Now we can set Edit's text.
  // code block 1
except
  // code block 2
end;
// ...
```

In all situations, when developing LCL applications, it is most important to know about this and to keep in mind that some floating point operations can go different way with different widgetsets. Then you can think of an appropriate

way to workaround this, but this should not go unnoticed.

Issues when moving from Windows to *nix etc

Issues specific to Linux, macOS, Android and other Unixes are described here. Not all subjects may apply to all platforms

On Unix there is no "application directory"

Many programmers are used to calling ExtractFilePath(ParamStr(0)) or Application.ExeName to get the location of the executable, and then search for the necessary files for the program execution (Images, XML files, database files, etc) based on the location of the executable. This is wrong on unixes. The string on ParamStr(0) may contain a directory other than the one of the executable, and it also varies between different shell programs (sh, bash, etc).

Even if Application.ExeName could in fact know the directory where the executable is, that file could be a symbolic link, so you could get the directory of the link instead (depending on the Linux kernel version, you either get the directory of the link or of the program binary itself).

To avoid this read the sections about configuration files and data files.

Making do without Windows COM Automation

With Windows, COM Automation is a powerful way not only of manipulating other programs remotely but also for allowing other programs to manipulate your program. With Delphi you can make your program both an COM Automation client and a COM Automation server, meaning it can both manipulate other programs and in turn be manipulated by other programs. For examples, see Using COM Automation to interact with OpenOffice and Microsoft Office.

macOS alternative

Unfortunately, COM Automation isn't available on macOS and Linux. However, you can simulate some of the functionality of COM Automation on macOS using AppleScript.

AppleScript is similar to COM Automation in some ways. For example, you can write scripts that manipulate other programs. Here's a very simple example of AppleScript that starts NeoOffice (the Mac version of OpenOffice.org):

```
tell application "NeoOffice"
  launch
end tell
```

An app that is designed to be manipulated by AppleScript provides a "dictionary" of classes and commands that can be used with the app, similar to the classes of a Windows Automation server. However, even apps like NeoOffice that don't provide a dictionary will still respond to the commands "launch", "activate" and "quit". AppleScript can be run from the macOS Script Editor or Finder or even converted to an app that you can drop on the dock just like any app. You can also run AppleScript from your program, as in this example:

```
fpsystem('myscript.applescript');
```

This assumes the script is in the indicated file. You can also run scripts on the fly from your app using the macOS OsaScript command:

```
fpsystem('osascript -e '#39'tell application "NeoOffice"'#39 +
      ' -e '#39'launch'#39' -e '#39'end tell'#39);
      {Note use of #39 to single-quote the parameters}
```

However, these examples are just the equivalent of the following Open command:

```
fpsystem('open -a NeoOffice');
```

Similarly, in macOS you can emulate the Windows shell commands to launch a web browser and launch an email client with:

```
fpsystem('open -a safari
"http://gigaset.com/shc/0,1935,hq_en_0_141387_rArNrNrNrN,00.html"');
```

and

```
fpsystem('open -a mail "mailto:ss4200@invalid.org"');
```

which assumes, fairly safely, that a macOS system will have the Safari and Mail applications installed. Of course, you should never make assumptions like this, and for the two previous examples, you can in fact just rely on macOS to do

the right thing and pick the user's default web browser and email client if you instead use these variations:

```
fpsystem('open "http://gigaset.com/shc/0,1935,hq_en_0_141387_rArNrNrNrN,00.html"');
```

and

```
fpsystem('open "mailto:ss4200@invalid.org"');
```

Do not forget to include the Unix unit in your uses clause if you use `fpsystem` or `shell` (interchangeable).

The real power of AppleScript is to manipulate programs remotely to create and open documents and automate other activities. How much you can do with a program depends on how extensive its AppleScript dictionary is (if it has one). For example, Microsoft's Office X programs are not very usable with AppleScript, whereas the newer Office 2004 programs have completely rewritten AppleScript dictionaries that compare in many ways with what's available via the Windows Office Automation servers.

Linux alternatives

While Linux shells support sophisticated command line scripting, the type of scripting is limited to what can be passed to a program on the command line. There is no single, unified way to access a program's internal classes and commands with Linux the way they are via Windows COM Automation and macOS AppleScript. However, individual desktop environments (GNOME/KDE) and application frameworks often provide such methods of interprocess communication. On GNOME see Bonobo Components. KDE has the KParts framework, DCOP. OpenOffice has a platform neutral API for controlling the office remotely (google OpenOffice SDK) - though you would probably have to write glue code in another language that has bindings (such as Python) to use it. In addition, some applications have "server modes" activated by special command-line options that allow them to be controlled from another process. It is also possible (Borland did it with Kylix document browser) to "embed" one top-level X application window into another using XReparentWindow (I think).

As with Windows, many macOS and Linux programs are made up of multiple library files (.dylib and .so extensions). Sometimes these libraries are designed so you can also use them in programs you write. While this can be a way of adding some of the functionality of an external program to your program, it's not really the same as running and manipulating the external program itself. Instead, your program is just linking to and using the external program's library similar to the way it would use any programming library.

Alternatives for Windows API functions

Many Windows programs use the Windows API extensively. In cross-platform applications Win API functions in the Windows unit should not be used, or should be enclosed by a conditional compile (e.g. {$IFDEF MSWINDOWS}).

Fortunately many of the commonly used Windows API functions are implemented in a multiplatform way in the unit lclintf. This can be a solution for programs which rely heavily on the Windows API, although the best solution is to replace these calls with true cross-platform components from the LCL. You can replace calls to GDI painting functions with calls to a TCanvas object's methods, for example.

Key codes

Fortunately, detecting key codes (e.g. on KeyUp events) is portable: see LCL Key Handling.

Installing your application

See Deploying Your Application.

See also

External links

- http://www.midnightbeach.com/KylixForDelphiProgrammers.html A guide for Windows programmers starting with Kylix. Many of concepts / code snippets apply to Lazarus.
- http://www.stack.nl/~marcov/porting.pdf A guide for writing portable source code, mainly between different compilers.