Lazarus/FPC Libraries

| **English (en)** | español (es) | français (fr) | 日本語 (ja) | русский (ru) |

This page describes the possibilities for creating libraries with Lazarus/FPC and using them in projects and packages.

Related Topics

- Creating bindings for C libraries - How to convert C header files (.h) to pascal units

General

Static linking

FPC compiles and links a static executable by default. That means it tells the linker to put all .o files of the project and all packages into one big executable.

- Advantages:
  - No external dependencies.
- Disadvantages:
  - No code is shared between different programs on the same computer.
  - You can not load/unload a plugin.

Dynamic libraries

The alternative to static linking is using dynamic libraries. The idea of dynamic libraries is to share code between programs,

- Advantages:
  - Saving the memory of the code
  - Reducing startup time for often used libraries
  - Allowing plugins.
- Disadvantages:
  - They are slower for seldomly used libs.
  - Their structure and internals are more complicated (this is mainly a problem for the compiler).
  - Their initialization is different (see below)
  - Sharing code requires a version system to only mix compatible code.

Operating Systems

Operating System	Dynamic library extension	Static library extension	Library prefix
FreeBSD	.so	.a	lib

macOS	.dylib	.a	lib
Linux	.so	.a	lib
Windows	.dll	.lib	n/a
Haiku	.so	.a	lib

The library prefix and extension columns indicate how the names of libraries are resolved and created for the listed operating systems.

FreeBSD

To Do.

macOS

Under macOS, the library name will always have the lib prefix when it is created. So if you create a dynamic library called test, this will result in the file `libtest.dylib`. When importing routines from shared libraries, it is not necessary to give the library prefix or the library filename extension.

Before FPC 2.2.2 released in August 2008, 32 bit .dylibs required exported symbols to be prefixed with an underscore. This is no longer the case.

The standard locations for dynamic libraries are `~/lib`, `/usr/local/lib`, and `/usr/lib`. Starting with macOS 10.15 (Catalina) `/usr/lib` is on the system read-only volume so cannot be used for third-party libraries. While you may also place the .dylib file in a non-standard location in your file system, you then must add that location to one of these environment variables:

- LD_LIBRARY_PATH
- DYLD_LIBRARY_PATH
- DYLD_FALLBACK_LIBRARY_PATH

Private shared libraries should be included in your application's Application Bundle. This is especially important for libraries which do not have a stable ABI (eg OpenSSL) so that they are revision-locked to the application and cannot be superseded by any other, even newer, versions that may be available to the operating system.

Refer to the See also section below for additional macOS information.

Linux

A dynamic library filename always has the form 'lib'+packagename+'.so'+version. For example: `libz.so.1` and `libz.so.1.2.2`.

Linux searches a library in this order:

- first in the paths of the environment variable `LD_LIBRARY_PATH`
- then in /lib
- then /usr/lib
- finally the paths of `/etc/ld.so.conf`

Using `ldconfig` after copying your library file into a lib directory may help with caching issues.

To share memory (`GetMem`/`FreeMem`, strings) with other libraries (not written in FPC) under Linux, you should use the **unit cmem**. This unit must be added as **the very first unit** in the uses section of the project main source file (typically .lpr), so that its initialization section is called before any other unit can allocate memory.

Windows

Windows searches a library in:

- first in the current directory
- then the system directory
- finally the directories specified in the environment variable PATH (system and user path).

to do: wasn't there a change in Vista+ behaviour for this?

When using an FPC made DLL with a non-FPC host application, exception handling can lead to some problems. Workarounds are mentioned in the #Initialization section.

ppumove, .ppu, .ppl

FPC normally creates for every unit a .ppu and .o file. The **.ppu file** contains every important information of the .pas/.pp file (types, required filenames like the .o file), while the **.o file** contains the assembler code and the mangled names understood by the current system.

The **ppumove** tool included with every FPC installation, converts one or several .ppu and .o files into a dynamic library. It does this by calling the linker to gather all .o files into a .so (windows: .dll) file and removes the .o filename references from the .ppu file. These new .ppu files are normally called **.ppl files.**

For example:

You have the output directory of a package (where the .ppu files are):

```
ppumove -o packagename -e ppl *.ppu
```

This will convert all .ppu files into .ppl files and creates a libpackagename.so (windows: packagename.dll). Note that under Linux the prefix 'lib' is always prepended.

This new library can already be used by other programming languages like C. Or by FPC programs by using the **external** modifiers. But the initialization/finalization sections must be called automatically. This includes the initialization/finalization of the heap manager. This means no strings or GetMem. Of course FPC programmers are spoiled and they can get more.

External - statically loading a dynamic library

The static loading of a dynamic library can be done in code, by using the external keyword, like in the example below which shows how a gtk function is loaded:

```
const
{$ifdef win32}
  gtklib = 'libgtk-win32-2.0-0.dll';
{$else}
  {$ifdef darwin}
    gtklib = 'gtk-x11-2.0';
    {$linklib gtk-x11-2.0}
  {$else}
    gtklib = 'libgtk-x11-2.0.so';
  {$endif}
{$endif}

procedure gtk_widget_set_events(widget:PGtkWidget; events:gint); cdecl;
external gtklib;
```

Loadlibrary - dynamically loading a dynamic library

Loading a dynamic library is simple with the Loadlibrary function of the unit dynlibs.

The main problem is to get the filename, which depends on the version and the operating system. Since 2.2.2, a constant "sharedsuffix" is declared in unit dynlibs to ease this. It maps to the relevant extension (dll/so/dylib) without a point as prefix.

Here are the functions declared in the unit dynlibs:

```
Function SafeLoadLibrary(Name : AnsiString) : TLibHandle;
Function LoadLibrary(Name : AnsiString) : TLibHandle;
Function GetProcedureAddress(Lib : TlibHandle; ProcName : AnsiString) :
Pointer;
Function UnloadLibrary(Lib : TLibHandle) : Boolean;
```

SafeLoadLibrary is the same as LoadLibrary but it also disables arithmetic exceptions which some libraries expect to be off on loading.

Pseudocode of the usage:

```
uses dynlibs;

var
  MyLibC: TLibHandle;
  MyProc: TMyProc;
begin
  MyLibC := LoadLibrary('libc.' + SharedSuffix);
  if MyLibC = 0 then Exit;
  MyProc := TMyProc(GetProcedureAddress(MyLibC, 'getpt');
  if MyProc = nil then Exit;
end;
```

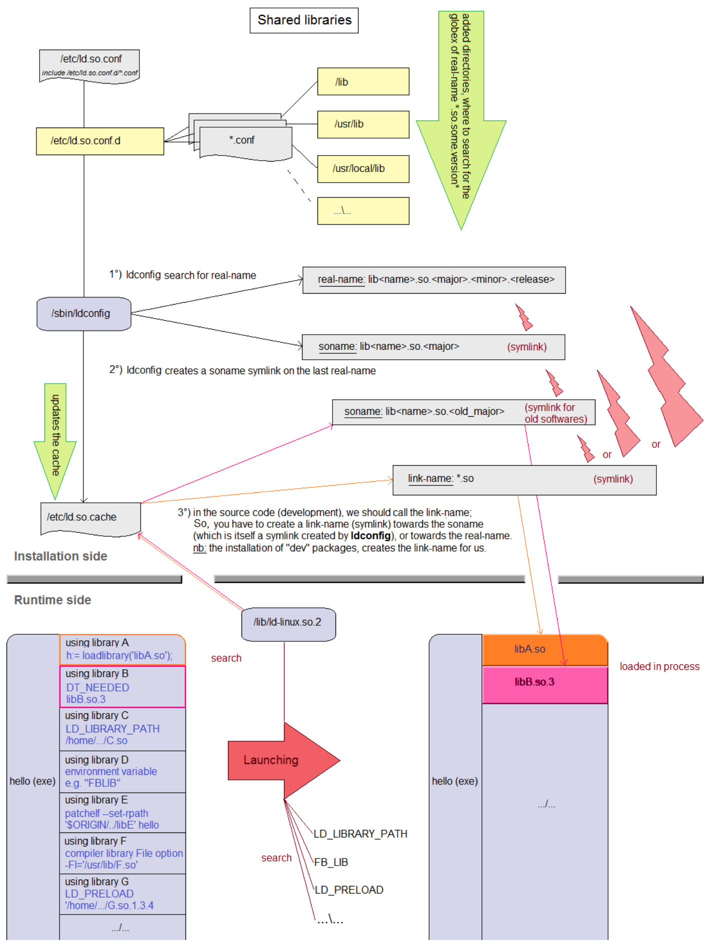
Pseudocode for using a function, taking parameters:

```
uses ...dynlibs...

procedure UseDLL;
type
  TMyFunc=function (aInt:Integer; aStr: string):String; StdCall;
var
  MyLibC: TLibHandle= dynlibs.NilHandle;
  MyFunc: TMyFunc;
  FuncResult: string;
begin
  MyLibC := LoadLibrary('libc.' + SharedSuffix);
  if MyLibC = dynlibs.NilHandle then Exit;  //DLL was not loaded successfully
  MyFunc:= TMyFunc(GetProcedureAddress(MyLibC, 'MyFunc');
  FuncResult:= MyFunc (5,'Test');  //Executes the function
  if MyLibC <>  DynLibs.NilHandle then if FreeLibrary(MyLibC) then MyLibC:=
DynLibs.NilHandle;  //Unload the lib, if already loaded
end;
```

Note: Before 1.9.4, the unit dl was used for this. Since 1.9.4, dynlibs provides a portable alternative. Note that pretty much any use of unit dl that can't be substituted by using dynlibs is typically already unportable amongst Unices. This alone makes it advisable to use unit dynlibs as much as possible.

Simplified overview of the system, when loading a shared library under Linux


Shared libraries
/etc/ld.so.conf
include /etc/ld.so.conf.d/*.conf
/etc/ld.so.conf.d
*.conf
/lib
/usr/lib
/usr/local/lib
...\...
added directories, where to search for the globex of real-name *.so.some.version*
1°) ldconfig search for real-name
/sbin/ldconfig
real-name: lib<name>.so.<major>.<minor>.<release>
soname: lib<name>.so.<major> (symlink)
2°) ldconfig creates a soname symlink on the last real-name
soname: lib<name>.so.<old_major> (symlink for old softwares)
or
or
link-name: *.so (symlink)
updates the cache
/etc/ld.so.cache
3°) in the source code (development), we should call the link-name; So, you have to create a link-name (symlink) towards the soname (which is itself a symlink created by ldconfig), or towards the real-name. nb: the installation of "dev" packages, creates the link-name for us.
Installation side
Runtime side
/lib/ld-linux.so.2
search
hello (exe)
using library A
h:= loadlibrary('libA.so');
using library B
DT_NEEDED
libB.so.3
using library C
LD_LIBRARY_PATH
/home/.../C.so
using library D
environment variable
e.g. "FBLIB"
using library E
patchelf --set-rpath
'$ORIGIN/../libE' hello
using library F
compiler library File option
-Fl='/usr/lib/F.so'
using library G
LD_PRELOAD
'/home/.../G.so.1.3.4
.../...
Launching
search
LD_LIBRARY_PATH
FB_LIB
LD_PRELOAD
...\...
hello (exe)
libA.so
libB.so.3
.../...
loaded in process

This chart summarizes how to configure a shared library (referenced by its soname, and its real-name, at least) in /etc/ld.so.conf and to update the library cache using *ldconfig* utility, to inform the Linux system library loader named *ldd* (other methods to load a shared library, with an environment variable like LD_LIBRARY_PATH, with code, by inclusion in the binary elf, etc, are also illustrated in the chart). For information, it's often necessary to check if a soft-symlink exists between the soname and the real-name, at least. The installation of development packages (*-dev*.deb files, for a Debian installation) always delivers such soft-symlink (ex.: abc.so) pointing to the soname (ex.: abc.so.{9}) on which the development package depends. Release packages sometimes forget this, whereas they are best placed to retrieve and to make point an old installation towards a new installed real-name library: indeed, shared libraries should respect the takeover of precedence in their existing APIs, features, at least inside the same soname (backward compatibility).

Creating a library

See the Free Pascal Documentation: http://www.freepascal.org/docs-html/prog/progse55.html

Initialization

Every unit can contain an initialization section. The order of the initialization sections depend on the uses sections of each unit.

The initialization of the RTL itself (in system.pp) that is done before your initialization sections are entered slightly differs between a program and a shared library. Most notably there is a difference in how the exception handling is initialized:

Exception handling

Windows

On Windows systems it has been reported that there is no exception handler installed to catch hardware exceptions like access violations or floating point errors; all other native Pascal exceptions that are raised with Raise will work however.

If you really need to catch these hardware exceptions you have to install your own handler, on Windows (beginning with versions XP and later) this is conveniently possible with AddVectoredExceptionHandler() and RemoveVectoredExceptionHandler(), there are some code examples in the above mentioned bug report. Note: work has been started in FPC 2.7.1/trunk to address this problem using 32 bit SEH.

UNIX and UNIX-like systems

On UNIX and UNIX-like systems (BSD, Darwin, Linux etc) no exception handler is installed to catch hardware exceptions (signals) like access violations in shared libraries. If you wish to catch these exceptions (signals) in a library code, you need to explicitly call `HookSignal(RTL_SIGDEFAULT)` during library initialization and call `UnhookSignal(RTL_SIGDEFAULT)` during finalization.

Keep in mind that when you enable signal handling in your shared library, it will replace signal handlers installed by a host application or other shared libraries of the current process. It may cause unexpected behavior. It is known that Java on Linux uses the SIGSEGV signal for its internal functionality and will crash if a third-party JNI library overrides the SIGSEGV signal handler.

FPU mask

For Delphi compatibility reasons, Free Pascal on i386 will unmask all FPU exceptions in the floating point unit (but without installing a handler for the dll, see above). This unmasking is done before your initialization section begins. If your DLL will be used with a host application that is made with a different compiler (almost all other compilers except FPC and Delphi/ C++ Builder *please verify C++ builder*) this will badly interfere with the host application's exception handling.

After the LoadLibrary() call, the host application won't be able to catch any floating point exceptions anymore. The FPU will suddenly have gained the ability to throw hardware exceptions (in the thread that called LoadLibrary()), and the host application will most likely not expect this to happen and not be able to handle it properly. The workaround is to put

```
Set8087CW(Get8087CW or $3f);
```

in the initialization section to mask them again for the thread that loaded the library. This won't have any unwanted side effects other than restoring what the host application had all the time. In your DLL you would not have been able to catch them anyway (at least not on Windows, see above). If you really need to catch zero-divide and related errors in a DLL that is written for a non-FPC application, you have to

- have a handler installed like mentioned above
- temporarily unmask the FPU before the try

- mask it again before you return to the host application

Finalization

Every unit can contain a finalization section. The order is the reverse order of the initialization sections.

After all finalization sections are executed it will call the procedure that you have previously assigned to Dll_Process_Detach_Hook.

Be very careful what you do in this procedure:

- all other units will already be finalized
- parts of the RTL might not work as expected anymore
- creating and freeing objects on the heap or even using Pascal strings inside this procedure is a sure way to call for trouble.

If you have to stop threads and free objects you must find a way to do this before the library unloading - it is too late to do it here.

Versions, Distribution

Libraries tend to grow and change over time. Adding new features is no problem, but removing a public method or changing its parameters makes the library incompatible. That means either an installed library (.so, .dll, .dylib) is replaced by a compatible one **or** a new library must be added to the system. That's why every library contains a version.

To do: explain how to set/use this version both on Windows and *nix

To load a dynamic library (**dlopen** of **unit dynlibs**) the correct filename must be known. Under Linux this means that you have to know the version number.

To do: proposal how the IDE should create version numbers

See also

- macOS Programming Tips - Libraries
- macOS Dynamic Libraries
- macOS Static Libraries
- FPC shared library
- FPC packages
- Translations / i18n / localizations for programs