NAME

perlos2 - Perl under OS/2, DOS, Win0.3*, Win0.95 and WinNT.

SYNOPSIS

One can read this document in the following formats:

```
man perlos2
view perl perlos2
explorer perlos2.html
info perlos2
```

to list some (not all may be available simultaneously), or it may be read *as is*: either as *README.os2*, or *pod/perlos2.pod*.

To read the *.INF* version of documentation (**very** recommended) outside of OS/2, one needs an IBM's reader (may be available on IBM ftp sites (?) (URL anyone?)) or shipped with PC DOS 7.0 and IBM's Visual Age C++ 3.5.

A copy of a Win* viewer is contained in the "Just add OS/2 Warp" package

```
ftp://ftp.software.ibm.com/ps/products/os2/tools/jaow/jaow.zip
```

in *?:\JUST_ADD\view.exe*. This gives one an access to EMX's *.INF* docs as well (text form is available in */emx/doc* in EMX's distribution). There is also a different viewer named xview.

Note that if you have *lynx.exe* or *netscape.exe* installed, you can follow WWW links from this document in *.INF* format. If you have EMX docs installed correctly, you can follow library links (you need to have `view emxbook` working by setting `EMXBOOK` environment variable as it is described in EMX docs).

DESCRIPTION

Target

The target is to make OS/2 one of the best supported platform for using/building/developing Perl and *Perl applications*, as well as make Perl the best language to use under OS/2. The secondary target is to try to make this work under DOS and Win* as well (but not **too** hard).

The current state is quite close to this target. Known limitations:

- Some *nix programs use fork() a lot; with the mostly useful flavors of perl for OS/2 (there are several built simultaneously) this is supported; but some flavors do not support this (e.g., when Perl is called from inside REXX). Using fork() after *use*ing dynamically loading extensions would not work with *very* old versions of EMX.
- You need a separate perl executable *perl__.exe* (see *perl__.exe*) if you want to use PM code in your application (as Perl/Tk or OpenGL Perl modules do) without having a text-mode window present.

  While using the standard *perl.exe* from a text-mode window is possible too, I have seen cases when this causes degradation of the system stability. Using *perl__.exe* avoids such a degradation.
- There is no simple way to access WPS objects. The only way I know is via `OS2::REXX` and `SOM` extensions (see *OS2::REXX*, *Som*). However, we do not have access to convenience methods of Object-REXX. (Is it possible at all? I know of no Object-REXX API.) The `SOM` extension (currently in alpha-text) may eventually remove this shortcoming; however, due to the fact that DII is not supported by the `SOM` module, using `SOM` is not as convenient as one would like it.

Please keep this list up-to-date by informing me about other items.

Other OSes

Since OS/2 port of perl uses a remarkable EMX environment, it can run (and build extensions, and - possibly - be built itself) under any environment which can run EMX. The current list is DOS, DOS-inside-OS/2, Win0.3*, Win0.95 and WinNT. Out of many perl flavors, only one works, see *perl_.exe*.

Note that not all features of Perl are available under these environments. This depends on the features the *extender* - most probably RSX - decided to implement.

Cf. *Prerequisites*.

Prerequisites

EMX

EMX runtime is required (may be substituted by RSX). Note that it is possible to make *perl_.exe* to run under DOS without any external support by binding *emx.exe*/*rsx.exe* to it, see *emxbind*. Note that under DOS for best results one should use RSX runtime, which has much more functions working (like `fork`, `popen` and so on). In fact RSX is required if there is no VCPI present. Note the RSX requires DPMI. Many implementations of DPMI are known to be very buggy, beware!

Only the latest runtime is supported, currently `0.9d fix 03`. Perl may run under earlier versions of EMX, but this is not tested.

One can get different parts of EMX from, say

```
http://www.leo.org/pub/comp/os/os2/leo/gnu/emx+gcc/
http://powerusersbbs.com/pub/os2/dev/   [EMX+GCC Development]
http://hobbes.nmsu.edu/pub/os2/dev/emx/v0.9d/
```

The runtime component should have the name *emxrt.zip*.

NOTE. When using *emx.exe*/*rsx.exe*, it is enough to have them on your path. One does not need to specify them explicitly (though this

```
emx perl_.exe -de 0
```

will work as well.)

RSX

To run Perl on DPMI platforms one needs RSX runtime. This is needed under DOS-inside-OS/2, Win0.3*, Win0.95 and WinNT (see *Other OSes*). RSX would not work with VCPI only, as EMX would, it requires DMPI.

Having RSX and the latest *sh.exe* one gets a fully functional ***nix**-ish environment under DOS, say, `fork`, `` `` `` and pipe-`open` work. In fact, MakeMaker works (for static build), so one can have Perl development environment under DOS.

One can get RSX from, say

```
ftp://ftp.cdrom.com/pub/os2/emx09c/contrib
ftp://ftp.uni-bielefeld.de/pub/systems/msdos/misc
ftp://ftp.leo.org/pub/comp/os/os2/leo/devtools/emx+gcc/contrib
```

Contact the author on `rainer@mathematik.uni-bielefeld.de`.

The latest *sh.exe* with DOS hooks is available in

```
http://www.ilyaz.org/software/os2/
```

as *sh_dos.zip* or under similar names starting with `sh`, `pdksh` etc.

HPFS

Perl does not care about file systems, but the perl library contains many files with long

names, so to install it intact one needs a file system which supports long file names.

Note that if you do not plan to build the perl itself, it may be possible to fool EMX to truncate file names. This is not supported, read EMX docs to see how to do it.

pdksh

To start external programs with complicated command lines (like with pipes in between, and/or quoting of arguments), Perl uses an external shell. With EMX port such shell should be named *sh.exe*, and located either in the wired-in-during-compile locations (usually *F:/bin*), or in configurable location (see *PERL_SH_DIR*).

For best results use EMX pdksh. The standard binary (5.2.14 or later) runs under DOS (with *RSX*) as well, see

```
http://www.ilyaz.org/software/os2/
```

Starting Perl programs under OS/2 (and DOS and...)

Start your Perl program *foo.pl* with arguments `arg1 arg2 arg3` the same way as on any other platform, by

```
perl foo.pl arg1 arg2 arg3
```

If you want to specify perl options `-my_opts` to the perl itself (as opposed to your program), use

```
perl -my_opts foo.pl arg1 arg2 arg3
```

Alternately, if you use OS/2-ish shell, like CMD or 4os2, put the following at the start of your perl script:

```
extproc perl -S -my_opts
```

rename your program to *foo.cmd*, and start it by typing

```
foo arg1 arg2 arg3
```

Note that because of stupid OS/2 limitations the full path of the perl script is not available when you use `extproc`, thus you are forced to use `-S` perl switch, and your script should be on the `PATH`. As a plus side, if you know a full path to your script, you may still start it with

```
perl ../../blah/foo.cmd arg1 arg2 arg3
```

(note that the argument `-my_opts` is taken care of by the `extproc` line in your script, see *`extproc` on the first line*).

To understand what the above *magic* does, read perl docs about `-S` switch - see *perlrun*, and cmdref about `extproc`:

```
view perl perlrun
man perlrun
view cmdref extproc
help extproc
```

or whatever method you prefer.

There are also endless possibilities to use *executable extensions* of 4os2, *associations* of WPS and so on... However, if you use *nixish shell (like *sh.exe* supplied in the binary distribution), you need to follow the syntax specified in *"Switches" in perlrun*.

Note that **-S** switch supports scripts with additional extensions *.cmd*, *.btm*, *.bat*, *.pl* as well.

Starting OS/2 (and DOS) programs under Perl

This is what system() (see *"system" in perlfunc*), `` ` `` (see *"I/O Operators" in perlop*), and *open pipe* (see *"open" in perlfunc*) are for. (Avoid exec() (see *"exec" in perlfunc*) unless you know what you do).

Note however that to use some of these operators you need to have a sh-syntax shell installed (see *Pdksh*, *Frequently asked questions*), and perl should be able to find it (see *PERL_SH_DIR*).

The cases when the shell is used are:

1 One-argument system() (see *"system" in perlfunc*), exec() (see *"exec" in perlfunc*) with redirection or shell meta-characters;

2 Pipe-open (see *"open" in perlfunc*) with the command which contains redirection or shell meta-characters;

3 Backticks `` ` `` (see *"I/O Operators" in perlop*) with the command which contains redirection or shell meta-characters;

4 If the executable called by system()/exec()/pipe-open()/`` ` `` is a script with the "magic" `#!` line or `extproc` line which specifies shell;

5 If the executable called by system()/exec()/pipe-open()/`` ` `` is a script without "magic" line, and `$ENV{EXECSHELL}` is set to shell;

6 If the executable called by system()/exec()/pipe-open()/`` ` `` is not found (is not this remark obsolete?);

7 For globbing (see *"glob" in perlfunc*, *"I/O Operators" in perlop*) (obsolete? Perl uses builtin globbing nowadays...).

For the sake of speed for a common case, in the above algorithms backslashes in the command name are not considered as shell metacharacters.

Perl starts scripts which begin with cookies `extproc` or `#!` directly, without an intervention of shell. Perl uses the same algorithm to find the executable as *pdksh*: if the path on `#!` line does not work, and contains `/`, then the directory part of the executable is ignored, and the executable is searched in `.` and on `PATH`. To find arguments for these scripts Perl uses a different algorithm than *pdksh*: up to 3 arguments are recognized, and trailing whitespace is stripped.

If a script does not contain such a cooky, then to avoid calling *sh.exe*, Perl uses the same algorithm as *pdksh*: if `$ENV{EXECSHELL}` is set, the script is given as the first argument to this command, if not set, then `$ENV{COMSPEC} /c` is used (or a hardwired guess if `$ENV{COMSPEC}` is not set).

When starting scripts directly, Perl uses exactly the same algorithm as for the search of script given by **-S** command-line option: it will look in the current directory, then on components of `$ENV{PATH}` using the following order of appended extensions: no extension, *.cmd*, *.btm*, *.bat*, *.pl*.

Note that Perl will start to look for scripts only if OS/2 cannot start the specified application, thus `system 'blah'` will not look for a script if there is an executable file *blah.exe anywhere* on `PATH`. In other words, `PATH` is essentially searched twice: once by the OS for an executable, then by Perl for scripts.

Note also that executable files on OS/2 can have an arbitrary extension, but *.exe* will be automatically appended if no dot is present in the name. The workaround is as simple as that: since *blah.* and *blah* denote the same file (at list on FAT and HPFS file systems), to start an executable residing in file *n:/bin/blah* (no extension) give an argument `n:/bin/blah.` (dot appended) to system().

Perl will start PM programs from VIO (=text-mode) Perl process in a separate PM session; the opposite is not true: when you start a non-PM program from a PM Perl process, Perl would not run it

in a separate session. If a separate session is desired, either ensure that shell will be used, as in `system 'cmd /c myprog'`, or start it using optional arguments to system() documented in `OS2::Process` module. This is considered to be a feature.

Frequently asked questions

"It does not work"

Perl binary distributions come with a *testperl.cmd* script which tries to detect common problems with misconfigured installations. There is a pretty large chance it will discover which step of the installation you managed to goof. `;-)`

I cannot run external programs

- Did you run your programs with `-w` switch? See *"2 (and DOS) programs under Perl" in Starting OS*.
- Do you try to run *internal* shell commands, like `` `copy a b` `` (internal for *cmd.exe*), or `` `glob a*b` `` (internal for ksh)? You need to specify your shell explicitly, like `` `cmd /c copy a b` ``, since Perl cannot deduce which commands are internal to your shell.

I cannot embed perl into my program, or use perl.dll from my program.

Is your program EMX-compiled with `-Zmt -Zcrtdll`?

Well, nowadays Perl DLL should be usable from a differently compiled program too... If you can run Perl code from REXX scripts (see *OS2::REXX*), then there are some other aspect of interaction which are overlooked by the current hackish code to support differently-compiled principal programs.

If everything else fails, you need to build a stand-alone DLL for perl. Contact me, I did it once. Sockets would not work, as a lot of other stuff.

Did you use *ExtUtils::Embed*?

Some time ago I had reports it does not work. Nowadays it is checked in the Perl test suite, so grep *./t* subdirectory of the build tree (as well as **.t* files in the *.//lib* subdirectory) to find how it should be done "correctly".

`` and pipe-open do not work under DOS.

This may a variant of just *I cannot run external programs*, or a deeper problem. Basically: you *need* RSX (see *Prerequisites*) for these commands to work, and you may need a port of *sh.exe* which understands command arguments. One of such ports is listed in *Prerequisites* under RSX. Do not forget to set variable *`PERL_SH_DIR`* as well.

DPMI is required for RSX.

Cannot start find.exe "pattern" file

The whole idea of the "standard C API to start applications" is that the forms `foo` and `"foo"` of program arguments are completely interchangable. *find* breaks this paradigm;

```
find "pattern" file
find pattern file
```

are not equivalent; *find* cannot be started directly using the above API. One needs a way to surround the doublequotes in some other quoting construction, necessarily having an extra non-Unixish shell in between.

Use one of

```
system 'cmd', '/c', 'find "pattern" file';
`cmd /c 'find "pattern" file'`
```

This would start *find.exe* via *cmd.exe* via `sh.exe` via `perl.exe`, but this is a price to pay if you want to use non-conforming program.

INSTALLATION

Automatic binary installation

The most convenient way of installing a binary distribution of perl is via perl installer *install.exe*. Just follow the instructions, and 99% of the installation blues would go away.

Note however, that you need to have *unzip.exe* on your path, and EMX environment *running*. The latter means that if you just installed EMX, and made all the needed changes to *Config.sys*, you may need to reboot in between. Check EMX runtime by running

```
emxrev
```

Binary installer also creates a folder on your desktop with some useful objects. If you need to change some aspects of the work of the binary installer, feel free to edit the file *Perl.pkg*. This may be useful e.g., if you need to run the installer many times and do not want to make many interactive changes in the GUI.

Things not taken care of by automatic binary installation:

`PERL_BADLANG`

: may be needed if you change your codepage *after* perl installation, and the new value is not supported by EMX. See *PERL_BADLANG*.

`PERL_BADFREE`

: see *PERL_BADFREE*.

Config.pm

: This file resides somewhere deep in the location you installed your perl library, find it out by

```
perl -MConfig -le "print $INC{'Config.pm'}"
```

: While most important values in this file *are* updated by the binary installer, some of them may need to be hand-edited. I know no such data, please keep me informed if you find one. Moreover, manual changes to the installed version may need to be accompanied by an edit of this file.

NOTE. Because of a typo the binary installer of 5.00305 would install a variable `PERL_SHPATH` into *Config.sys*. Please remove this variable and put *`PERL_SH_DIR`* instead.

Manual binary installation

As of version 5.00305, OS/2 perl binary distribution comes split into 11 components. Unfortunately, to enable configurable binary installation, the file paths in the zip files are not absolute, but relative to some directory.

Note that the extraction with the stored paths is still necessary (default with unzip, specify `-d` to pkunzip). However, you need to know where to extract the files. You need also to manually change entries in *Config.sys* to reflect where did you put the files. Note that if you have some primitive unzipper (like `pkunzip`), you may get a lot of warnings/errors during unzipping. Upgrade to `(w)unzip`.

Below is the sample of what to do to reproduce the configuration on my machine. In *VIEW.EXE* you can press `Ctrl-Insert` now, and cut-and-paste from the resulting file - created in the directory you started *VIEW.EXE* from.

For each component, we mention environment variables related to each installation directory. Either

choose directories to match your values of the variables, or create/append-to variables to take into account the directories.

Perl VIO and PM executables (dynamically linked)

```
unzip perl_exc.zip *.exe *.ico -d f:/emx.add/bin
unzip perl_exc.zip *.dll -d f:/emx.add/dll
```

(have the directories with `*.exe` on PATH, and `*.dll` on LIBPATH);

Perl_ VIO executable (statically linked)

```
unzip perl_aou.zip -d f:/emx.add/bin
```

(have the directory on PATH);

Executables for Perl utilities

```
unzip perl_utl.zip -d f:/emx.add/bin
```

(have the directory on PATH);

Main Perl library

```
unzip perl_mlb.zip -d f:/perllib/lib
```

If this directory is exactly the same as the prefix which was compiled into *perl.exe*, you do not need to change anything. However, for perl to find the library if you use a different path, you need to `set PERLLIB_PREFIX` in *Config.sys*, see *PERLLIB_PREFIX*.

Additional Perl modules

```
unzip perl_ste.zip -d f:/perllib/lib/site_perl/5.10.1/
```

Same remark as above applies. Additionally, if this directory is not one of directories on @INC (and @INC is influenced by `PERLLIB_PREFIX`), you need to put this directory and subdirectory *./os2* in `PERLLIB` or `PERL5LIB` variable. Do not use `PERL5LIB` unless you have it set already. See *"ENVIRONMENT" in perl*.

[Check whether this extraction directory is still applicable with the new directory structure layout!]

Tools to compile Perl modules

```
unzip perl_blb.zip -d f:/perllib/lib
```

Same remark as for *perl_ste.zip*.

Manpages for Perl and utilities

```
unzip perl_man.zip -d f:/perllib/man
```

This directory should better be on `MANPATH`. You need to have a working *man* to access these files.

Manpages for Perl modules

```
unzip perl_mam.zip -d f:/perllib/man
```

This directory should better be on `MANPATH`. You need to have a working man to access these files.

Source for Perl documentation

```
unzip perl_pod.zip -d f:/perllib/lib
```

This is used by the `perldoc` program (see *perldoc*), and may be used to generate HTML documentation usable by WWW browsers, and documentation in zillions of other formats: `info`, `LaTeX`, `Acrobat`, `FrameMaker` and so on. [Use programs such as *pod2latex* etc.]

Perl manual in *.INF* format

```
unzip perl_inf.zip -d d:/os2/book
```

This directory should better be on `BOOKSHELF`.

Pdksh

```
unzip perl_sh.zip -d f:/bin
```

This is used by perl to run external commands which explicitly require shell, like the commands using *redirection* and *shell metacharacters*. It is also used instead of explicit */bin/sh*.

Set `PERL_SH_DIR` (see *PERL_SH_DIR*) if you move *sh.exe* from the above location.

Note. It may be possible to use some other sh-compatible shell (untested).

After you installed the components you needed and updated the *Config.sys* correspondingly, you need to hand-edit *Config.pm*. This file resides somewhere deep in the location you installed your perl library, find it out by

```
perl -MConfig -le "print $INC{'Config.pm'}"
```

You need to correct all the entries which look like file paths (they currently start with `f:/`).

Warning

The automatic and manual perl installation leave precompiled paths inside perl executables. While these paths are overwriteable (see *PERLLIB_PREFIX*, *PERL_SH_DIR*), some people may prefer binary editing of paths inside the executables/DLLs.

Accessing documentation

Depending on how you built/installed perl you may have (otherwise identical) Perl documentation in the following formats:

OS/2 .INF file

Most probably the most convenient form. Under OS/2 view it as

```
view perl
view perl perlfunc
view perl less
view perl ExtUtils::MakeMaker
```

(currently the last two may hit a wrong location, but this may improve soon). Under Win* see *SYNOPSIS*.

If you want to build the docs yourself, and have *OS/2 toolkit*, run

```
pod2ipf > perl.ipf
```

in */perllib/lib/pod* directory, then

```
ipfc /inf perl.ipf
```

(Expect a lot of errors during the both steps.) Now move it on your BOOKSHELF path.

Plain text

If you have perl documentation in the source form, perl utilities installed, and GNU groff installed, you may use

```
perldoc perlfunc
perldoc less
perldoc ExtUtils::MakeMaker
```

to access the perl documentation in the text form (note that you may get better results using perl manpages).

Alternately, try running pod2text on *.pod* files.

Manpages

If you have *man* installed on your system, and you installed perl manpages, use something like this:

```
man perlfunc
man 3 less
man ExtUtils.MakeMaker
```

to access documentation for different components of Perl. Start with

```
man perl
```

Note that dot (.) is used as a package separator for documentation for packages, and as usual, sometimes you need to give the section - 3 above - to avoid shadowing by the *less(1) manpage*.

Make sure that the directory **above** the directory with manpages is on our `MANPATH`, like this

```
  set MANPATH=c:/man;f:/perllib/man
```

for Perl manpages in `f:/perllib/man/man1/` etc.

HTML

If you have some WWW browser available, installed the Perl documentation in the source form, and Perl utilities, you can build HTML docs. Cd to directory with *.pod* files, and do like this

```
cd f:/perllib/lib/pod
pod2html
```

After this you can direct your browser the file *perl.html* in this directory, and go ahead with reading docs, like this:

```
explore file:///f:/perllib/lib/pod/perl.html
```

Alternatively you may be able to get these docs prebuilt from CPAN.

GNU info files

Users of Emacs would appreciate it very much, especially with `CPerl` mode loaded. You need to get latest `pod2texi` from `CPAN`, or, alternately, the prebuilt info pages.

PDF files

for `Acrobat` are available on CPAN (may be for slightly older version of perl).

LaTeX docs

can be constructed using `pod2latex`.

BUILD

Here we discuss how to build Perl under OS/2. There is an alternative (but maybe older) view on *http://www.shadow.net/~troc/os2perl.html*.

The short story

Assume that you are a seasoned porter, so are sure that all the necessary tools are already present on your system, and you know how to get the Perl source distribution. Untar it, change to the extract directory, and

```
gnupatch -p0 < os2\diff.configure
sh Configure -des -D prefix=f:/perllib
make
make test
make install
make aout_test
make aout_install
```

This puts the executables in f:/perllib/bin. Manually move them to the `PATH`, manually move the built *perl*.dll* to `LIBPATH` (here for Perl DLL * is a not-very-meaningful hex checksum), and run

```
make installcmd INSTALLCMDDIR=d:/ir/on/path
```

Assuming that the `man`-files were put on an appropriate location, this completes the installation of minimal Perl system. (The binary distribution contains also a lot of additional modules, and the documentation in INF format.)

What follows is a detailed guide through these steps.

Prerequisites

You need to have the latest EMX development environment, the full GNU tool suite (gawk renamed to awk, and GNU *find.exe* earlier on path than the OS/2 *find.exe*, same with *sort.exe*, to check use

```
find --version
sort --version
```

). You need the latest version of *pdksh* installed as *sh.exe*.

Check that you have **BSD** libraries and headers installed, and - optionally - Berkeley DB headers and libraries, and crypt.

Possible locations to get the files:

```
ftp://hobbes.nmsu.edu/os2/unix/
ftp://ftp.cdrom.com/pub/os2/unix/
ftp://ftp.cdrom.com/pub/os2/dev32/
ftp://ftp.cdrom.com/pub/os2/emx09c/
```

It is reported that the following archives contain enough utils to build perl: *gnufutil.zip*, *gnusutil.zip*, *gnututil.zip*, *gnused.zip*, *gnupatch.zip*, *gnuawk.zip*, *gnumake.zip*, *gnugrep.zip*, *bsddev.zip* and *ksh527rt.zip* (or a later version). Note that all these utilities are known to be available from LEO:

```
ftp://ftp.leo.org/pub/comp/os/os2/leo/gnu
```

Note also that the *db.lib* and *db.a* from the EMX distribution are not suitable for multi-threaded

compile (even single-threaded flavor of Perl uses multi-threaded C RTL, for compatibility with XFree86-OS/2). Get a corrected one from

```
http://www.ilyaz.org/software/os2/db_mt.zip
```

If you have *exactly the same version of Perl* installed already, make sure that no copies or perl are currently running. Later steps of the build may fail since an older version of *perl.dll* loaded into memory may be found. Running `make test` becomes meaningless, since the test are checking a previous build of perl (this situation is detected and reported by *lib/os2_base.t* test). Do not forget to unset `PERL_EMXLOAD_SEC` in environment.

Also make sure that you have */tmp* directory on the current drive, and `.` directory in your `LIBPATH`. One may try to correct the latter condition by

```
set BEGINLIBPATH .\.
```

if you use something like *CMD.EXE* or latest versions of *4os2.exe*. (Setting BEGINLIBPATH to just `.` is ignored by the OS/2 kernel.)

Make sure your gcc is good for `-Zomf` linking: run `omflibs` script in */emx/lib* directory.

Check that you have link386 installed. It comes standard with OS/2, but may be not installed due to customization. If typing

```
link386
```

shows you do not have it, do *Selective install*, and choose `Link object modules` in *Optional system utilities/More*. If you get into link386 prompts, press `Ctrl-C` to exit.

Getting perl source

You need to fetch the latest perl source (including developers releases). With some probability it is located in

```
http://www.cpan.org/src/5.0
http://www.cpan.org/src/5.0/unsupported
```

If not, you may need to dig in the indices to find it in the directory of the current maintainer.

Quick cycle of developers release may break the OS/2 build time to time, looking into

```
http://www.cpan.org/ports/os2/
```

may indicate the latest release which was publicly released by the maintainer. Note that the release may include some additional patches to apply to the current source of perl.

Extract it like this

```
tar vzxf perl5.00409.tar.gz
```

You may see a message about errors while extracting *Configure*. This is because there is a conflict with a similarly-named file *configure*.

Change to the directory of extraction.

Application of the patches

You need to apply the patches in *./os2/diff.** like this:

```
gnupatch -p0 < os2\diff.configure
```

You may also need to apply the patches supplied with the binary distribution of perl. It also makes sense to look on the perl5-porters mailing list for the latest OS/2-related patches (see *http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/*). Such patches usually contain strings `/os2/` and `patch`, so it makes sense looking for these strings.

Hand-editing

You may look into the file *./hints/os2.sh* and correct anything wrong you find there. I do not expect it is needed anywhere.

Making

```
sh Configure -des -D prefix=f:/perllib
```

`prefix` means: where to install the resulting perl library. Giving correct prefix you may avoid the need to specify `PERLLIB_PREFIX`, see *PERLLIB_PREFIX*.

Ignore the message about missing `ln`, and about `-c` option to tr. The latter is most probably already fixed, if you see it and can trace where the latter spurious warning comes from, please inform me.

Now

```
make
```

At some moment the built may die, reporting a *version mismatch* or *unable to run perl*. This means that you do not have `.` in your LIBPATH, so *perl.exe* cannot find the needed *perl67B2.dll* (treat these hex digits as line noise). After this is fixed the build should finish without a lot of fuss.

Testing

Now run

```
make test
```

All tests should succeed (with some of them skipped). If you have the same version of Perl installed, it is crucial that you have `.` early in your LIBPATH (or in BEGINLIBPATH), otherwise your tests will most probably test the wrong version of Perl.

Some tests may generate extra messages similar to

A lot of `bad free`

> in database tests related to Berkeley DB. *This should be fixed already.* If it persists, you may disable this warnings, see *PERL_BADFREE*.

Process terminated by SIGTERM/SIGINT

> This is a standard message issued by OS/2 applications. *nix applications die in silence. It is considered to be a feature. One can easily disable this by appropriate sighandlers.
>
> However the test engine bleeds these message to screen in unexpected moments. Two messages of this kind *should* be present during testing.

To get finer test reports, call

```
perl t/harness
```

The report with *io/pipe.t* failing may look like this:

```
Failed Test  Status Wstat Total Fail  Failed  List of failed
-------------------------------------------------------------
io/pipe.t                    12    1   8.33%  9
7 tests skipped, plus 56 subtests skipped.
```

```
  Failed 1/195 test scripts, 99.49% okay. 1/6542 subtests failed, 99.98%
okay.
```

The reasons for most important skipped tests are:

op/fs.t

18 Checks `atime` and `mtime` of `stat()` - unfortunately, HPFS provides only 2sec time granularity (for compatibility with FAT?).

25 Checks `truncate()` on a filehandle just opened for write - I do not know why this should or should not work.

op/stat.t

Checks `stat()`. Tests:

4 Checks `atime` and `mtime` of `stat()` - unfortunately, HPFS provides only 2sec time granularity (for compatibility with FAT?).

Installing the built perl

If you haven't yet moved `perl*.dll` onto LIBPATH, do it now.

Run

```
  make install
```

It would put the generated files into needed locations. Manually put *perl.exe*, *perl__.exe* and *perl___.exe* to a location on your PATH, *perl.dll* to a location on your LIBPATH.

Run

```
  make installcmd INSTALLCMDDIR=d:/ir/on/path
```

to convert perl utilities to *.cmd* files and put them on PATH. You need to put *.EXE*-utilities on path manually. They are installed in `$prefix/bin`, here `$prefix` is what you gave to *Configure*, see *Making*.

If you use `man`, either move the installed **/man/* directories to your `MANPATH`, or modify `MANPATH` to match the location. (One could have avoided this by providing a correct `manpath` option to *./Configure*, or editing *./config.sh* between configuring and making steps.)

a.out-style build

Proceed as above, but make *perl_.exe* (see *perl_.exe*) by

```
  make perl_
```

test and install by

```
  make aout_test
  make aout_install
```

Manually put *perl_.exe* to a location on your PATH.

Note. The build process for `perl_` *does not know* about all the dependencies, so you should make sure that anything is up-to-date, say, by doing

```
  make perl_dll
```

first.

Building a binary distribution

[This section provides a short overview only...]

Building should proceed differently depending on whether the version of perl you install is already present and used on your system, or is a new version not yet used. The description below assumes that the version is new, so installing its DLLs and *.pm* files will not disrupt the operation of your system even if some intermediate steps are not yet fully working.

The other cases require a little bit more convoluted procedures. Below I suppose that the current version of Perl is `5.8.2`, so the executables are named accordingly.

1. Fully build and test the Perl distribution. Make sure that no tests are failing with `test` and `aout_test` targets; fix the bugs in Perl and the Perl test suite detected by these tests. Make sure that `all_test` make target runs as clean as possible. Check that `os2/perlrexx.cmd` runs fine.

2. Fully install Perl, including `installcmd` target. Copy the generated DLLs to `LIBPATH`; copy the numbered Perl executables (as in *perl5.8.2.exe*) to `PATH`; copy `perl_.exe` to `PATH` as `perl_5.8.2.exe`. Think whether you need backward-compatibility DLLs. In most cases you do not need to install them yet; but sometime this may simplify the following steps.

3. Make sure that `CPAN.pm` can download files from CPAN. If not, you may need to manually install `Net::FTP`.

4. Install the bundle `Bundle::OS2_default`

   ```
   perl5.8.2 -MCPAN -e "install Bundle::OS2_default" < nul |& tee
   00cpan_i_1
   ```

   This may take a couple of hours on 1GHz processor (when run the first time). And this should not be necessarily a smooth procedure. Some modules may not specify required dependencies, so one may need to repeat this procedure several times until the results stabilize.

   ```
   perl5.8.2 -MCPAN -e "install Bundle::OS2_default" < nul |& tee
   00cpan_i_2
   perl5.8.2 -MCPAN -e "install Bundle::OS2_default" < nul |& tee
   00cpan_i_3
   ```

   Even after they stabilize, some tests may fail.

   Fix as many discovered bugs as possible. Document all the bugs which are not fixed, and all the failures with unknown reasons. Inspect the produced logs *00cpan_i_1* to find suspiciously skipped tests, and other fishy events.

   Keep in mind that *installation* of some modules may fail too: for example, the DLLs to update may be already loaded by *CPAN.pm*. Inspect the `install` logs (in the example above *00cpan_i_1* etc) for errors, and install things manually, as in

   ```
   cd $CPANHOME/.cpan/build/Digest-MD5-2.31
   make install
   ```

   Some distributions may fail some tests, but you may want to install them anyway (as above, or via `force install` command of `CPAN.pm` shell-mode).

   Since this procedure may take quite a long time to complete, it makes sense to "freeze" your CPAN configuration by disabling periodic updates of the local copy of CPAN index: set `index_expire` to some big value (I use 365), then save the settings

   ```
   CPAN> o conf index_expire 365
   CPAN> o conf commit
   ```

Reset back to the default value `1` when you are finished.

5. When satisfied with the results, rerun the `installcmd` target. Now you can copy `perl5.8.2.exe` to `perl.exe`, and install the other OMF-build executables: `perl__.exe` etc. They are ready to be used.

6. Change to the `./pod` directory of the build tree, download the Perl logo *CamelGrayBig.BMP*, and run

   ```
   ( perl2ipf > perl.ipf ) |& tee 00ipf
   ipfc /INF perl.ipf |& tee 00inf
   ```

   This produces the Perl docs online book `perl.INF`. Install in on `BOOKSHELF` path.

7. Now is the time to build statically linked executable *perl_.exe* which includes newly-installed via `Bundle::OS2_default` modules. Doing testing via `CPAN.pm` is going to be painfully slow, since it statically links a new executable per XS extension.

   Here is a possible workaround: create a toplevel *Makefile.PL* in *$CPANHOME/.cpan/build/* with contents being (compare with *Making executables with a custom collection of statically loaded extensions*)

   ```
   use ExtUtils::MakeMaker;
   WriteMakefile NAME => 'dummy';
   ```

   execute this as

   ```
   perl_5.8.2.exe Makefile.PL <nul |& tee 00aout_c1
   make -k all test <nul |& 00aout_t1
   ```

   Again, this procedure should not be absolutely smooth. Some `Makefile.PL`'s in subdirectories may be buggy, and would not run as "child" scripts. The interdependency of modules can strike you; however, since non-XS modules are already installed, the prerequisites of most modules have a very good chance to be present.

   If you discover some glitches, move directories of problematic modules to a different location; if these modules are non-XS modules, you may just ignore them - they are already installed; the remaining, XS, modules you need to install manually one by one.

   After each such removal you need to rerun the `Makefile.PL`/`make` process; usually this procedure converges soon. (But be sure to convert all the necessary external C libraries from *.lib* format to *.a* format: run one of

   ```
   emxaout foo.lib
   emximp -o foo.a foo.lib
   ```

   whichever is appropriate.) Also, make sure that the DLLs for external libraries are usable with with executables compiled without `-Zmtd` options.

   When you are sure that only a few subdirectories lead to failures, you may want to add `-j4` option to `make` to speed up skipping subdirectories with already finished build.

   When you are satisfied with the results of tests, install the build C libraries for extensions:

   ```
   make install |& tee 00aout_i
   ```

   Now you can rename the file *./perl.exe* generated during the last phase to *perl_5.8.2.exe*; place it on `PATH`; if there is an inter-dependency between some XS modules, you may need to repeat the `test`/`install` loop with this new executable and some excluded modules - until the procedure converges.

   Now you have all the necessary *.a* libraries for these Perl modules in the places where Perl builder can find it. Use the perl builder: change to an empty directory, create a "dummy" *Makefile.PL* again, and run

```
perl_5.8.2.exe Makefile.PL |& tee 00c
make perl       |& tee 00p
```

This should create an executable *./perl.exe* with all the statically loaded extensions built in. Compare the generated *perlmain.c* files to make sure that during the iterations the number of loaded extensions only increases. Rename *./perl.exe* to *perl_5.8.2.exe* on `PATH`.

When it converges, you got a functional variant of *perl_5.8.2.exe*; copy it to `perl_.exe`. You are done with generation of the local Perl installation.

8. Make sure that the installed modules are actually installed in the location of the new Perl, and are not inherited from entries of @INC given for inheritance from the older versions of Perl: set `PERLLIB_582_PREFIX` to redirect the new version of Perl to a new location, and copy the installed files to this new location. Redo the tests to make sure that the versions of modules inherited from older versions of Perl are not needed.

   Actually, the log output of *pod2ipf* during the step 6 gives a very detailed info about which modules are loaded from which place; so you may use it as an additional verification tool.

   Check that some temporary files did not make into the perl install tree. Run something like this

   ```
   pfind . -f
   "!(/\.(pm|pl|ix|al|h|a|lib|txt|pod|imp|bs|dll|ld|bs|inc|xbm|yml|cgi|u
   u|e2x|skip|packlist|eg|cfg|html|pub|enc|all|ini|po|pot)$/i or
   /^\w+$/") | less
   ```

   in the install tree (both top one and *sitelib* one).

   Compress all the DLLs with *lxlite*. The tiny *.exe* can be compressed with `/c:max` (the bug only appears when there is a fixup in the last 6 bytes of a page (?); since the tiny executables are much smaller than a page, the bug will not hit). Do not compress `perl_.exe` - it would not work under DOS.

9. Now you can generate the binary distribution. This is done by running the test of the CPAN distribution `OS2::SoftInstaller`. Tune up the file *test.pl* to suit the layout of current version of Perl first. Do not forget to pack the necessary external DLLs accordingly. Include the description of the bugs and test suite failures you could not fix. Include the small-stack versions of Perl executables from Perl build directory.

   Include *perl5.def* so that people can relink the perl DLL preserving the binary compatibility, or can create compatibility DLLs. Include the diff files (`diff -pu old new`) of fixes you did so that people can rebuild your version. Include *perl5.map* so that one can use remote debugging.

10. Share what you did with the other people. Relax. Enjoy fruits of your work.

11. Brace yourself for thanks, bug reports, hate mail and spam coming as result of the previous step. No good deed should remain unpunished!

Building custom .EXE files

The Perl executables can be easily rebuilt at any moment. Moreover, one can use the *embedding* interface (see *perlembed*) to make very customized executables.

Making executables with a custom collection of statically loaded extensions

It is a little bit easier to do so while *decreasing* the list of statically loaded extensions. We discuss this case only here.

1. Change to an empty directory, and create a placeholder <Makefile.PL>:

   ```
   use ExtUtils::MakeMaker;
   WriteMakefile NAME => 'dummy';
   ```

2. Run it with the flavor of Perl (*perl.exe* or *perl_.exe*) you want to rebuild.

```
perl_ Makefile.PL
```

3. Ask it to create new Perl executable:

```
make perl
```

(you may need to manually add `PERLTYPE=-DPERL_CORE` to this commandline on some versions of Perl; the symptom is that the command-line globbing does not work from OS/2 shells with the newly-compiled executable; check with

```
.\perl.exe -wle "print for @ARGV" *
```

).

4. The previous step created *perlmain.c* which contains a list of newXS() calls near the end. Removing unnecessary calls, and rerunning

```
make perl
```

will produce a customized executable.

Making executables with a custom search-paths

The default perl executable is flexible enough to support most usages. However, one may want something yet more flexible; for example, one may want to find Perl DLL relatively to the location of the EXE file; or one may want to ignore the environment when setting the Perl-library search patch, etc.

If you fill comfortable with *embedding* interface (see *perlembed*), such things are easy to do repeating the steps outlined in *Making executables with a custom collection of statically loaded extensions*, and doing more comprehensive edits to main() of *perlmain.c*. The people with little desire to understand Perl can just rename main(), and do necessary modification in a custom main() which calls the renamed function in appropriate time.

However, there is a third way: perl DLL exports the main() function and several callbacks to customize the search path. Below is a complete example of a "Perl loader" which

1. Looks for Perl DLL in the directory `$exedir/../dll`;
2. Prepends the above directory to `BEGINLIBPATH`;
3. Fails if the Perl DLL found via `BEGINLIBPATH` is different from what was loaded on step 1; e.g., another process could have loaded it from `LIBPATH` or from a different value of `BEGINLIBPATH`. In these cases one needs to modify the setting of the system so that this other process either does not run, or loads the DLL from `BEGINLIBPATH` with `LIBPATHSTRICT=T` (available with kernels after September 2000).
4. Loads Perl library from `$exedir/../dll/lib/`.
5. Uses Bourne shell from `$exedir/../dll/sh/ksh.exe`.

For best results compile the C file below with the same options as the Perl DLL. However, a lot of functionality will work even if the executable is not an EMX applications, e.g., if compiled with

```
  gcc -Wall -DDOSISH -DOS2=1 -O2 -s -Zomf -Zsys perl-starter.c
-DPERL_DLL_BASENAME=\"perl312F\" -Zstack 8192 -Zlinker /PM:VIO
```

Here is the sample C file:

```
  #define INCL_DOS
```

```
  #define INCL_NOPM
  /* These are needed for compile if os2.h includes os2tk.h, not os2emx.h
*/
  #define INCL_DOSPROCESS
  #include <os2.h>

  #include "EXTERN.h"
  #define PERL_IN_MINIPERLMAIN_C
  #include "perl.h"

  static char *me;
  HMODULE handle;

  static void
  die_with(char *msg1, char *msg2, char *msg3, char *msg4)
  {
     ULONG c;
     char *s = " error: ";

     DosWrite(2, me, strlen(me), &c);
     DosWrite(2, s, strlen(s), &c);
     DosWrite(2, msg1, strlen(msg1), &c);
     DosWrite(2, msg2, strlen(msg2), &c);
     DosWrite(2, msg3, strlen(msg3), &c);
     DosWrite(2, msg4, strlen(msg4), &c);
     DosWrite(2, "\r\n", 2, &c);
     exit(255);
  }

  typedef ULONG (*fill_extLibpath_t)(int type, char *pre, char *post, int
replace, char *msg);
  typedef int (*main_t)(int type, char *argv[], char *env[]);
  typedef int (*handler_t)(void* data, int which);

  #ifndef PERL_DLL_BASENAME
  #  define PERL_DLL_BASENAME "perl"
  #endif

  static HMODULE
  load_perl_dll(char *basename)
  {
      char buf[300], fail[260];
      STRLEN l, dirl;
      fill_extLibpath_t f;
      ULONG rc_fullname;
      HMODULE handle, handle1;

      if (_execname(buf, sizeof(buf) - 13) != 0)
          die_with("Can't find full path: ", strerror(errno), "", "");
      /* XXXX Fill `me' with new value */
      l = strlen(buf);
      while (l && buf[l-1] != '/' && buf[l-1] != '\\')
          l--;
```

```
      dirl = l - 1;
      strcpy(buf + l, basename);
      l += strlen(basename);
      strcpy(buf + l, ".dll");
      if ( (rc_fullname = DosLoadModule(fail, sizeof fail, buf, &handle))
!= 0
           && DosLoadModule(fail, sizeof fail, basename, &handle) != 0 )
          die_with("Can't load DLL ", buf, "", "");
      if (rc_fullname)
          return handle;  /* was loaded with short name; all is fine */
      if (DosQueryProcAddr(handle, 0, "fill_extLibpath", (PFN*)&f))
          die_with(buf, ": DLL exports no symbol ", "fill_extLibpath", "");
      buf[dirl] = 0;
      if (f(0 /*BEGINLIBPATH*/, buf /* prepend */, NULL /* append */,
            0 /* keep old value */, me))
          die_with(me, ": prepending BEGINLIBPATH", "", "");
      if (DosLoadModule(fail, sizeof fail, basename, &handle1) != 0)
          die_with(me, ": finding perl DLL again via BEGINLIBPATH", "",
"");
      buf[dirl] = '\\';
      if (handle1 != handle) {
          if (DosQueryModuleName(handle1, sizeof(fail), fail))
              strcpy(fail, "???");
          die_with(buf, ":\n\tperl DLL via BEGINLIBPATH is different:
\n\t",
                   fail,
                   "\n\tYou may need to manipulate global BEGINLIBPATH and
LIBPATHSTRICT"
                   "\n\tso that the other copy is loaded via
BEGINLIBPATH.");
      }
      return handle;
  }

  int
  main(int argc, char **argv, char **env)
  {
      main_t f;
      handler_t h;

      me = argv[0];
      /**/
      handle = load_perl_dll(PERL_DLL_BASENAME);

      if (DosQueryProcAddr(handle, 0, "Perl_OS2_handler_install",
(PFN*)&h))
          die_with(PERL_DLL_BASENAME, ": DLL exports no symbol ",
"Perl_OS2_handler_install", "");
      if ( !h((void *)"~installprefix", Perlos2_handler_perllib_from)
           || !h((void *)"~dll", Perlos2_handler_perllib_to)
           || !h((void *)"~dll/sh/ksh.exe", Perlos2_handler_perl_sh) )
          die_with(PERL_DLL_BASENAME, ": Can't install @INC manglers", "",
"");

      if (DosQueryProcAddr(handle, 0, "dll_perlmain", (PFN*)&f))
```

```
            die_with(PERL_DLL_BASENAME, ": DLL exports no symbol ",
"dll_perlmain", "");
      return f(argc, argv, env);
  }
```

Build FAQ

Some / became \ in pdksh.

You have a very old pdksh. See *Prerequisites*.

'errno' - unresolved external

You do not have MT-safe *db.lib*. See *Prerequisites*.

Problems with tr or sed

reported with very old version of tr and sed.

Some problem (forget which ;-)

You have an older version of *perl.dll* on your LIBPATH, which broke the build of extensions.

Library ... not found

You did not run `omflibs`. See *Prerequisites*.

Segfault in make

You use an old version of GNU make. See *Prerequisites*.

op/sprintf test failure

This can result from a bug in emx sprintf which was fixed in 0.9d fix 03.

Specific (mis)features of OS/2 port

setpriority, getpriority

Note that these functions are compatible with *nix, not with the older ports of '94 - 95. The priorities are absolute, go from 32 to -95, lower is quicker. 0 is the default priority.

WARNING. Calling `getpriority` on a non-existing process could lock the system before Warp3 fixpak22. Starting with Warp3, Perl will use a workaround: it aborts getpriority() if the process is not present. This is not possible on older versions `2.*`, and has a race condition anyway.

system()

Multi-argument form of `system()` allows an additional numeric argument. The meaning of this argument is described in *OS2::Process*.

When finding a program to run, Perl first asks the OS to look for executables on `PATH` (OS/2 adds extension *.exe* if no extension is present). If not found, it looks for a script with possible extensions added in this order: no extension, *.cmd*, *.btm*, *.bat*, *.pl*. If found, Perl checks the start of the file for magic strings `"#!"` and `"extproc "`. If found, Perl uses the rest of the first line as the beginning of the command line to run this script. The only mangling done to the first line is extraction of arguments (currently up to 3), and ignoring of the path-part of the "interpreter" name if it can't be found using the full path.

E.g., `system 'foo', 'bar', 'baz'` may lead Perl to finding *C:/emx/bin/foo.cmd* with the first line being

```
 extproc /bin/bash    -x   -c
```

If */bin/bash.exe* is not found, then Perl looks for an executable *bash.exe* on `PATH`. If found in *C:/emx.add/bin/bash.exe*, then the above system() is translated to

```
system qw(C:/emx.add/bin/bash.exe -x -c C:/emx/bin/foo.cmd bar baz)
```

One additional translation is performed: instead of */bin/sh* Perl uses the hardwired-or-customized shell (see *PERL_SH_DIR*).

The above search for "interpreter" is recursive: if *bash* executable is not found, but *bash.btm* is found, Perl will investigate its first line etc. The only hardwired limit on the recursion depth is implicit: there is a limit 4 on the number of additional arguments inserted before the actual arguments given to system(). In particular, if no additional arguments are specified on the "magic" first lines, then the limit on the depth is 4.

If Perl finds that the found executable is of PM type when the current session is not, it will start the new process in a separate session of necessary type. Call via `OS2::Process` to disable this magic.

WARNING. Due to the described logic, you need to explicitly specify *.com* extension if needed. Moreover, if the executable *perl5.6.1* is requested, Perl will not look for *perl5.6.1.exe*. [This may change in the future.]

extproc on the first line

If the first chars of a Perl script are `"extproc "`, this line is treated as `#!`-line, thus all the switches on this line are processed (twice if script was started via cmd.exe). See *"DESCRIPTION" in perlrun*.

Additional modules:

OS2::Process, *OS2::DLL*, *OS2::REXX*, *OS2::PrfDB*, *OS2::ExtAttr*. These modules provide access to additional numeric argument for `system` and to the information about the running process, to DLLs having functions with REXX signature and to the REXX runtime, to OS/2 databases in the *.INI* format, and to Extended Attributes.

Two additional extensions by Andreas Kaiser, `OS2::UPM`, and `OS2::FTP`, are included into `ILYAZ` directory, mirrored on CPAN. Other OS/2-related extensions are available too.

Prebuilt methods:

`File::Copy::syscopy`
: used by `File::Copy::copy`, see *File::Copy*.

`DynaLoader::mod2fname`
: used by `DynaLoader` for DLL name mangling.

`Cwd::current_drive()`
: Self explanatory.

`Cwd::sys_chdir(name)`
: leaves drive as it is.

`Cwd::change_drive(name)`
: chanes the "current" drive.

`Cwd::sys_is_absolute(name)`
: means has drive letter and is_rooted.

`Cwd::sys_is_rooted(name)`
: means has leading `[/\\]` (maybe after a drive-letter:).

`Cwd::sys_is_relative(name)`
: means changes with current dir.

`Cwd::sys_cwd(name)`

Interface to cwd from EMX. Used by `Cwd::cwd`.

`Cwd::sys_abspath(name, dir)`

Really really odious function to implement. Returns absolute name of file which would have `name` if CWD were `dir`. `Dir` defaults to the current dir.

`Cwd::extLibpath([type])`

Get current value of extended library search path. If `type` is present and positive, works with `END_LIBPATH`, if negative, works with `LIBPATHSTRICT`, otherwise with `BEGIN_LIBPATH`.

`Cwd::extLibpath_set( path [, type ] )`

Set current value of extended library search path. If `type` is present and positive, works with <END_LIBPATH>, if negative, works with `LIBPATHSTRICT`, otherwise with `BEGIN_LIBPATH`.

`OS2::Error(do_harderror,do_exception)`

Returns `undef` if it was not called yet, otherwise bit 1 is set if on the previous call do_harderror was enabled, bit 2 is set if on previous call do_exception was enabled.

This function enables/disables error popups associated with hardware errors (Disk not ready etc.) and software exceptions.

I know of no way to find out the state of popups *before* the first call to this function.

`OS2::Errors2Drive(drive)`

Returns `undef` if it was not called yet, otherwise return false if errors were not requested to be written to a hard drive, or the drive letter if this was requested.

This function may redirect error popups associated with hardware errors (Disk not ready etc.) and software exceptions to the file POPUPLOG.OS2 at the root directory of the specified drive. Overrides OS2::Error() specified by individual programs. Given argument undef will disable redirection.

Has global effect, persists after the application exits.

I know of no way to find out the state of redirection of popups to the disk *before* the first call to this function.

OS2::SysInfo()

Returns a hash with system information. The keys of the hash are

```
MAX_PATH_LENGTH, MAX_TEXT_SESSIONS, MAX_PM_SESSIONS,
MAX_VDM_SESSIONS, BOOT_DRIVE, DYN_PRI_VARIATION,
MAX_WAIT, MIN_SLICE, MAX_SLICE, PAGE_SIZE,
VERSION_MAJOR, VERSION_MINOR, VERSION_REVISION,
MS_COUNT, TIME_LOW, TIME_HIGH, TOTPHYSMEM, TOTRESMEM,
TOTAVAILMEM, MAXPRMEM, MAXSHMEM, TIMER_INTERVAL,
MAX_COMP_LENGTH, FOREGROUND_FS_SESSION,
FOREGROUND_PROCESS
```

OS2::BootDrive()

Returns a letter without colon.

`OS2::MorphPM(serve)`, `OS2::UnMorphPM(serve)`

Transforms the current application into a PM application and back. The argument true means that a real message loop is going to be served. OS2::MorphPM() returns the PM message queue handle as an integer.

See *Centralized management of resources* for additional details.

`OS2::Serve_Messages(force)`

Fake on-demand retrieval of outstanding PM messages. If `force` is false, will not dispatch messages if a real message loop is known to be present. Returns number of messages retrieved.

Dies with "QUITing..." if WM_QUIT message is obtained.

`OS2::Process_Messages(force [, cnt])`

Retrieval of PM messages until window creation/destruction. If `force` is false, will not dispatch messages if a real message loop is known to be present.

Returns change in number of windows. If `cnt` is given, it is incremented by the number of messages retrieved.

Dies with "QUITing..." if WM_QUIT message is obtained.

`OS2::_control87(new,mask)`

the same as *_control87(3)* of EMX. Takes integers as arguments, returns the previous coprocessor control word as an integer. Only bits in `new` which are present in `mask` are changed in the control word.

OS2::get_control87()

gets the coprocessor control word as an integer.

`OS2::set_control87_em(new=MCW_EM,mask=MCW_EM)`

The variant of OS2::_control87() with default values good for handling exception mask: if no `mask`, uses exception mask part of `new` only. If no `new`, disables all the floating point exceptions.

See *Misfeatures* for details.

`OS2::DLLname([how [, \&xsub]])`

Gives the information about the Perl DLL or the DLL containing the C function bound to by `&xsub`. The meaning of `how` is: default (2): full name; 0: handle; 1: module name.

(Note that some of these may be moved to different libraries - eventually).

Prebuilt variables:

$OS2::emx_rev

numeric value is the same as _emx_rev of EMX, a string value the same as _emx_vprt (similar to `0.9c`).

$OS2::emx_env

same as _emx_env of EMX, a number similar to 0x8001.

$OS2::os_ver

a number `OS_MAJOR + 0.001 * OS_MINOR`.

$OS2::is_aout

true if the Perl library was compiled in AOUT format.

$OS2::can_fork

true if the current executable is an AOUT EMX executable, so Perl can fork. Do not use this, use the portable check for $Config::Config{dfork}.

$OS2::nsyserror

This variable (default is 1) controls whether to enforce the contents of $^E to start with `SYS0003`-like id. If set to 0, then the string value of $^E is what is available from the OS/2 message file. (Some messages in this file have an `SYS0003`-like id prepended, some not.)

Misfeatures

- Since *flock(3)* is present in EMX, but is not functional, it is emulated by perl. To disable the emulations, set environment variable `USE_PERL_FLOCK=0`.
- Here is the list of things which may be "broken" on EMX (from EMX docs):
  - The functions *recvmsg(3)*, *sendmsg(3)*, and *socketpair(3)* are not implemented.
  - *sock_init(3)* is not required and not implemented.
  - *flock(3)* is not yet implemented (dummy function). (Perl has a workaround.)
  - *kill(3)*: Special treatment of PID=0, PID=1 and PID=-1 is not implemented.
  - *waitpid(3)*:

    ```
    WUNTRACED
     Not implemented.
    waitpid() is not implemented for negative values of PID.
    ```

  Note that `kill -9` does not work with the current version of EMX.
- See *Text-mode filehandles*.
- Unix-domain sockets on OS/2 live in a pseudo-file-system `/sockets/...`. To avoid a failure to create a socket with a name of a different form, `"/socket/"` is prepended to the socket name (unless it starts with this already).

  This may lead to problems later in case the socket is accessed via the "usual" file-system calls using the "initial" name.
- Apparently, IBM used a compiler (for some period of time around '95?) which changes FP mask right and left. This is not *that* bad for IBM's programs, but the same compiler was used for DLLs which are used with general-purpose applications. When these DLLs are used, the state of floating-point flags in the application is not predictable.

  What is much worse, some DLLs change the floating point flags when in _DLLInitTerm() (e.g., *TCP32IP*). This means that even if you do not *call* any function in the DLL, just the act of loading this DLL will reset your flags. What is worse, the same compiler was used to compile some HOOK DLLs. Given that HOOK dlls are executed in the context of *all* the applications in the system, this means a complete unpredictablity of floating point flags on systems using such HOOK DLLs. E.g., *GAMESRVR.DLL* of **DIVE** origin changes the floating point flags on each write to the TTY of a VIO (windowed text-mode) applications.

  Some other (not completely debugged) situations when FP flags change include some video drivers (?), and some operations related to creation of the windows. People who code **OpenGL** may have more experience on this.

  Perl is generally used in the situation when all the floating-point exceptions are ignored, as is the default under EMX. If they are not ignored, some benign Perl programs would get a `SIGFPE` and would die a horrible death.

  To circumvent this, Perl uses two hacks. They help against *one* type of damage only: FP flags changed when loading a DLL.

  One of the hacks is to disable floating point exceptions on Perl startup (as is the default with EMX). This helps only with compile-time-linked DLLs changing the flags before main() had a chance to be called.

  The other hack is to restore FP flags after a call to dlopen(). This helps against similar damage done by DLLs _DLLInitTerm() at runtime. Currently no way to switch these hacks off is provided.

Modifications

Perl modifies some standard C library calls in the following ways:

`popen`

`my_popen` uses *sh.exe* if shell is required, cf. *PERL_SH_DIR*.

`tmpnam`

is created using `TMP` or `TEMP` environment variable, via `tempnam`.

`tmpfile`

If the current directory is not writable, file is created using modified `tmpnam`, so there may be a race condition.

`ctermid`

a dummy implementation.

`stat`

`os2_stat` special-cases */dev/tty* and */dev/con*.

`mkdir`, `rmdir`

these EMX functions do not work if the path contains a trailing `/`. Perl contains a workaround for this.

`flock`

Since *flock(3)* is present in EMX, but is not functional, it is emulated by perl. To disable the emulations, set environment variable `USE_PERL_FLOCK=0`.

Identifying DLLs

All the DLLs built with the current versions of Perl have ID strings identifying the name of the extension, its version, and the version of Perl required for this DLL. Run `bldlevel DLL-name` to find this info.

Centralized management of resources

Since to call certain OS/2 API one needs to have a correctly initialized `Win` subsystem, OS/2-specific extensions may require getting `HAB`s and `HMQ`s. If an extension would do it on its own, another extension could fail to initialize.

Perl provides a centralized management of these resources:

`HAB`

To get the HAB, the extension should call `hab = perl_hab_GET()` in C. After this call is performed, `hab` may be accessed as `Perl_hab`. There is no need to release the HAB after it is used.

If by some reasons *perl.h* cannot be included, use

```
extern int Perl_hab_GET(void);
```

instead.

`HMQ`

There are two cases:

- the extension needs an `HMQ` only because some API will not work otherwise. Use `serve = 0` below.
- the extension needs an `HMQ` since it wants to engage in a PM event loop. Use `serve = 1` below.

To get an HMQ, the extension should call hmq = perl_hmq_GET(serve) in C. After this call is performed, hmq may be accessed as Perl_hmq.

To signal to Perl that HMQ is not needed any more, call perl_hmq_UNSET(serve). Perl process will automatically morph/unmorph itself into/from a PM process if HMQ is needed/not-needed. Perl will automatically enable/disable WM_QUIT message during shutdown if the message queue is served/not-served.

NOTE. If during a shutdown there is a message queue which did not disable WM_QUIT, and which did not process the received WM_QUIT message, the shutdown will be automatically cancelled. Do not call perl_hmq_GET(1) unless you are going to process messages on an orderly basis.

* Treating errors reported by OS/2 API

There are two principal conventions (it is useful to call them Dos* and Win* - though this part of the function signature is not always determined by the name of the API) of reporting the error conditions of OS/2 API. Most of Dos* APIs report the error code as the result of the call (so 0 means success, and there are many types of errors). Most of Win* API report success/fail via the result being TRUE/FALSE; to find the reason for the failure one should call WinGetLastError() API.

Some Win* entry points also overload a "meaningful" return value with the error indicator; having a 0 return value indicates an error. Yet some other Win* entry points overload things even more, and 0 return value may mean a successful call returning a valid value 0, as well as an error condition; in the case of a 0 return value one should call WinGetLastError() API to distinguish a successful call from a failing one.

By convention, all the calls to OS/2 API should indicate their failures by resetting $^E. All the Perl-accessible functions which call OS/2 API may be broken into two classes: some die()s when an API error is encountered, the other report the error via a false return value (of course, this does not concern Perl-accessible functions which *expect* a failure of the OS/2 API call, having some workarounds coded).

Obviously, in the situation of the last type of the signature of an OS/2 API, it is must more convenient for the users if the failure is indicated by die()ing: one does not need to check $^E to know that something went wrong. If, however, this solution is not desirable by some reason, the code in question should reset $^E to 0 before making this OS/2 API call, so that the caller of this Perl-accessible function has a chance to distinguish a success-but-0-return value from a failure. (One may return undef as an alternative way of reporting an error.)

The macros to simplify this type of error propagation are

CheckOSError(expr)

Returns true on error, sets $^E. Expects expr() be a call of Dos*-style API.

CheckWinError(expr)

Returns true on error, sets $^E. Expects expr() be a call of Win*-style API.

SaveWinError(expr)

Returns expr, sets $^E from WinGetLastError() if expr is false.

SaveCroakWinError(expr,die,name1,name2)

Returns expr, sets $^E from WinGetLastError() if expr is false, and die()s if die and $^E are true. The message to die is the concatenated strings name1 and name2, separated by ": " from the contents of $^E.

WinError_2_Perl_rc

Sets Perl_rc to the return value of WinGetLastError().

FillWinError

Sets `Perl_rc` to the return value of WinGetLastError(), and sets $^E to the corresponding value.

`FillOSError(rc)`

Sets `Perl_rc` to `rc`, and sets $^E to the corresponding value.

* Loading DLLs and ordinals in DLLs

Some DLLs are only present in some versions of OS/2, or in some configurations of OS/2. Some exported entry points are present only in DLLs shipped with some versions of OS/2. If these DLLs and entry points were linked directly for a Perl executable/DLL or from a Perl extensions, this binary would work only with the specified versions/setups. Even if these entry points were not needed, the *load* of the executable (or DLL) would fail.

For example, many newer useful APIs are not present in OS/2 v2; many PM-related APIs require DLLs not available on floppy-boot setup.

To make these calls fail *only when the calls are executed*, one should call these API via a dynamic linking API. There is a subsystem in Perl to simplify such type of calls. A large number of entry points available for such linking is provided (see `entries_ordinals` - and also `PMWIN_entries` - in *os2ish.h*). These ordinals can be accessed via the APIs:

```
CallORD(), DeclFuncByORD(), DeclVoidFuncByORD(),
DeclOSFuncByORD(), DeclWinFuncByORD(), AssignFuncPByORD(),
DeclWinFuncByORD_CACHE(), DeclWinFuncByORD_CACHE_survive(),
DeclWinFuncByORD_CACHE_resetError_survive(),
DeclWinFunc_CACHE(), DeclWinFunc_CACHE_resetError(),
DeclWinFunc_CACHE_survive(), DeclWinFunc_CACHE_resetError_survive()
```

See the header files and the C code in the supplied OS/2-related modules for the details on usage of these functions.

Some of these functions also combine dynaloading semantic with the error-propagation semantic discussed above.

Perl flavors

Because of idiosyncrasies of OS/2 one cannot have all the eggs in the same basket (though EMX environment tries hard to overcome this limitations, so the situation may somehow improve). There are 4 executables for Perl provided by the distribution:

perl.exe

The main workhorse. This is a chimera executable: it is compiled as an `a.out`-style executable, but is linked with `omf`-style dynamic library *perl.dll*, and with dynamic CRT DLL. This executable is a VIO application.

It can load perl dynamic extensions, and it can fork().

Note. Keep in mind that fork() is needed to open a pipe to yourself.

perl_.exe

This is a statically linked `a.out`-style executable. It cannot load dynamic Perl extensions. The executable supplied in binary distributions has a lot of extensions prebuilt, thus the above restriction is important only if you use custom-built extensions. This executable is a VIO application.

This is the only executable with does not require OS/2. The friends locked into `M$` world would appreciate the fact that this executable runs under DOS, Win0.3*, Win0.95 and WinNT with an appropriate extender. See *Other OSes*.

perl__.exe

This is the same executable as *perl___.exe*, but it is a PM application.

Note. Usually (unless explicitly redirected during the startup) STDIN, STDERR, and STDOUT of a PM application are redirected to *nul*. However, it is possible to *see* them if you start `perl__.exe` from a PM program which emulates a console window, like *Shell mode* of Emacs or EPM. Thus it *is possible* to use Perl debugger (see *perldebug*) to debug your PM application (but beware of the message loop lockups - this will not work if you have a message queue to serve, unless you hook the serving into the getc() function of the debugger).

Another way to see the output of a PM program is to run it as

```
pm_prog args 2>&1 | cat -
```

with a shell *different* from *cmd.exe*, so that it does not create a link between a VIO session and the session of `pm_porg`. (Such a link closes the VIO window.) E.g., this works with *sh.exe* - or with Perl!

```
open P, 'pm_prog args 2>&1 |' or die;
print while <P>;
```

The flavor *perl__.exe* is required if you want to start your program without a VIO window present, but not `detached` (run `help detach` for more info). Very useful for extensions which use PM, like `Perl/Tk` or `OpenGL`.

Note also that the differences between PM and VIO executables are only in the *default* behaviour. One can start *any* executable in *any* kind of session by using the arguments `/fs`, `/pm` or `/win` switches of the command `start` (of *CMD.EXE* or a similar shell). Alternatively, one can use the numeric first argument of the `system` Perl function (see *OS2::Process*).

perl___.exe

This is an `omf`-style executable which is dynamically linked to *perl.dll* and CRT DLL. I know no advantages of this executable over `perl.exe`, but it cannot fork() at all. Well, one advantage is that the build process is not so convoluted as with `perl.exe`.

It is a VIO application.

Why strange names?

Since Perl processes the `#!`-line (cf. *"DESCRIPTION" in perlrun*, *"Switches" in perlrun*, *"Not a perl script" in perldiag*, *"No Perl script found in input" in perldiag*), it should know when a program *is a Perl*. There is some naming convention which allows Perl to distinguish correct lines from wrong ones. The above names are almost the only names allowed by this convention which do not contain digits (which have absolutely different semantics).

Why dynamic linking?

Well, having several executables dynamically linked to the same huge library has its advantages, but this would not substantiate the additional work to make it compile. The reason is the complicated-to-developers but very quick and convenient-to-users "hard" dynamic linking used by OS/2.

There are two distinctive features of the dyna-linking model of OS/2: first, all the references to external functions are resolved at the compile time; second, there is no runtime fixup of the DLLs after they are loaded into memory. The first feature is an enormous advantage over other models: it avoids conflicts when several DLLs used by an application export entries with the same name. In such cases "other" models of dyna-linking just choose between these two entry points using some random criterion - with predictable disasters as results. But it is the second feature which requires the build of *perl.dll*.

The address tables of DLLs are patched only once, when they are loaded. The addresses of the entry

points into DLLs are guaranteed to be the same for all the programs which use the same DLL. This removes the runtime fixup - once DLL is loaded, its code is read-only.

While this allows some (significant?) performance advantages, this makes life much harder for developers, since the above scheme makes it impossible for a DLL to be "linked" to a symbol in the *.EXE* file. Indeed, this would need a DLL to have different relocations tables for the (different) executables which use this DLL.

However, a dynamically loaded Perl extension is forced to use some symbols from the perl executable, e.g., to know how to find the arguments to the functions: the arguments live on the perl internal evaluation stack. The solution is to put the main code of the interpreter into a DLL, and make the *.EXE* file which just loads this DLL into memory and supplies command-arguments. The extension DLL cannot link to symbols in *.EXE*, but it has no problem linking to symbols in the *.DLL*.

This *greatly* increases the load time for the application (as well as complexity of the compilation). Since interpreter is in a DLL, the C RTL is basically forced to reside in a DLL as well (otherwise extensions would not be able to use CRT). There are some advantages if you use different flavors of perl, such as running *perl.exe* and *perl__.exe* simultaneously: they share the memory of *perl.dll*.

NOTE. There is one additional effect which makes DLLs more wasteful: DLLs are loaded in the shared memory region, which is a scarse resource given the 512M barrier of the "standard" OS/2 virtual memory. The code of *.EXE* files is also shared by all the processes which use the particular *.EXE*, but they are "shared in the private address space of the process"; this is possible because the address at which different sections of the *.EXE* file are loaded is decided at compile-time, thus all the processes have these sections loaded at same addresses, and no fixup of internal links inside the *.EXE* is needed.

Since DLLs may be loaded at run time, to have the same mechanism for DLLs one needs to have the address range of *any of the loaded* DLLs in the system to be available *in all the processes* which did not load a particular DLL yet. This is why the DLLs are mapped to the shared memory region.

Why chimera build?

Current EMX environment does not allow DLLs compiled using Unixish `a.out` format to export symbols for data (or at least some types of data). This forces `omf`-style compile of *perl.dll*.

Current EMX environment does not allow *.EXE* files compiled in `omf` format to fork(). fork() is needed for exactly three Perl operations:

- explicit fork() in the script,
- `open FH, "|-"`
- `open FH, "-|"`, in other words, opening pipes to itself.

While these operations are not questions of life and death, they are needed for a lot of useful scripts. This forces `a.out`-style compile of *perl.exe*.

ENVIRONMENT

Here we list environment variables with are either OS/2- and DOS- and Win*-specific, or are more important under OS/2 than under other OSes.

PERLLIB_PREFIX

Specific for EMX port. Should have the form

```
path1;path2
```

or

```
path1 path2
```

If the beginning of some prebuilt path matches *path1*, it is substituted with *path2*.

Should be used if the perl library is moved from the default location in preference to `PERL(5)LIB`, since this would not leave wrong entries in @INC. For example, if the compiled version of perl looks for @INC in *f:/perllib/lib*, and you want to install the library in *h:/opt/gnu*, do

```
set PERLLIB_PREFIX=f:/perllib/lib;h:/opt/gnu
```

This will cause Perl with the prebuilt @INC of

```
f:/perllib/lib/5.00553/os2
f:/perllib/lib/5.00553
f:/perllib/lib/site_perl/5.00553/os2
f:/perllib/lib/site_perl/5.00553
.
```

to use the following @INC:

```
h:/opt/gnu/5.00553/os2
h:/opt/gnu/5.00553
h:/opt/gnu/site_perl/5.00553/os2
h:/opt/gnu/site_perl/5.00553
.
```

PERL_BADLANG

If 0, perl ignores setlocale() failing. May be useful with some strange *locale*s.

PERL_BADFREE

If 0, perl would not warn of in case of unwarranted free(). With older perls this might be useful in conjunction with the module DB_File, which was buggy when dynamically linked and OMF-built.

Should not be set with newer Perls, since this may hide some *real* problems.

PERL_SH_DIR

Specific for EMX port. Gives the directory part of the location for *sh.exe*.

USE_PERL_FLOCK

Specific for EMX port. Since *flock(3)* is present in EMX, but is not functional, it is emulated by perl. To disable the emulations, set environment variable `USE_PERL_FLOCK=0`.

TMP or TEMP

Specific for EMX port. Used as storage place for temporary files.

Evolution

Here we list major changes which could make you by surprise.

Text-mode filehandles

Starting from version 5.8, Perl uses a builtin translation layer for text-mode files. This replaces the efficient well-tested EMX layer by some code which should be best characterized as a "quick hack".

In addition to possible bugs and an inability to follow changes to the translation policy with off/on switches of TERMIO translation, this introduces a serious incompatible change: before sysread() on text-mode filehandles would go through the translation layer, now it would not.

Priorities

`setpriority` and `getpriority` are not compatible with earlier ports by Andreas Kaiser. See `"setpriority, getpriority"`.

DLL name mangling: pre 5.6.2

With the release 5.003_01 the dynamically loadable libraries should be rebuilt when a different version of Perl is compiled. In particular, DLLs (including *perl.dll*) are now created with the names which contain a checksum, thus allowing workaround for OS/2 scheme of caching DLLs.

It may be possible to code a simple workaround which would

- find the old DLLs looking through the old @INC;
- mangle the names according to the scheme of new perl and copy the DLLs to these names;
- edit the internal `LX` tables of DLL to reflect the change of the name (probably not needed for Perl extension DLLs, since the internally coded names are not used for "specific" DLLs, they used only for "global" DLLs).
- edit the internal `IMPORT` tables and change the name of the "old" *perl????.dll* to the "new" *perl????.dll*.

DLL name mangling: 5.6.2 and beyond

In fact mangling of *extension* DLLs was done due to misunderstanding of the OS/2 dynaloading model. OS/2 (effectively) maintains two different tables of loaded DLL:

Global DLLs

: those loaded by the base name from `LIBPATH`; including those associated at link time;

specific DLLs

: loaded by the full name.

When resolving a request for a global DLL, the table of already-loaded specific DLLs is (effectively) ignored; moreover, specific DLLs are *always* loaded from the prescribed path.

There is/was a minor twist which makes this scheme fragile: what to do with DLLs loaded from

`BEGINLIBPATH` and `ENDLIBPATH`

: (which depend on the process)

. from `LIBPATH`

: which *effectively* depends on the process (although `LIBPATH` is the same for all the processes).

Unless `LIBPATHSTRICT` is set to `T` (and the kernel is after 2000/09/01), such DLLs are considered to be global. When loading a global DLL it is first looked in the table of already-loaded global DLLs. Because of this the fact that one executable loaded a DLL from `BEGINLIBPATH` and `ENDLIBPATH`, or . from `LIBPATH` may affect *which* DLL is loaded when *another* executable requests a DLL with the same name. *This* is the reason for version-specific mangling of the DLL name for perl DLL.

Since the Perl extension DLLs are always loaded with the full path, there is no need to mangle their names in a version-specific ways: their directory already reflects the corresponding version of perl, and @INC takes into account binary compatibility with older version. Starting from `5.6.2` the name mangling scheme is fixed to be the same as for Perl 5.005_53 (same as in a popular binary release). Thus new Perls will be able to *resolve the names* of old extension DLLs if @INC allows finding their directories.

However, this still does not guarantee that these DLL may be loaded. The reason is the mangling of the name of the *Perl DLL*. And since the extension DLLs link with the Perl DLL, extension DLLs for older versions would load an older Perl DLL, and would most probably segfault (since the data in this DLL is not properly initialized).

There is a partial workaround (which can be made complete with newer OS/2 kernels): create a

forwarder DLL with the same name as the DLL of the older version of Perl, which forwards the entry points to the newer Perl's DLL. Make this DLL accessible on (say) the `BEGINLIBPATH` of the new Perl executable. When the new executable accesses old Perl's extension DLLs, they would request the old Perl's DLL by name, get the forwarder instead, so effectively will link with the currently running (new) Perl DLL.

This may break in two ways:

- Old perl executable is started when a new executable is running has loaded an extension compiled for the old executable (ouph!). In this case the old executable will get a forwarder DLL instead of the old perl DLL, so would link with the new perl DLL. While not directly fatal, it will behave the same as new executable. This beats the whole purpose of explicitly starting an old executable.
- A new executable loads an extension compiled for the old executable when an old perl executable is running. In this case the extension will not pick up the forwarder - with fatal results.

With support for `LIBPATHSTRICT` this may be circumvented - unless one of DLLs is started from *.* from `LIBPATH` (I do not know whether `LIBPATHSTRICT` affects this case).

REMARK. Unless newer kernels allow *.* in `BEGINLIBPATH` (older do not), this mess cannot be completely cleaned. (It turns out that as of the beginning of 2002, *.* is not allowed, but *.\.* is - and it has the same effect.)

REMARK. `LIBPATHSTRICT`, `BEGINLIBPATH` and `ENDLIBPATH` are not environment variables, although *cmd.exe* emulates them on `SET ...` lines. From Perl they may be accessed by *Cwd::extLibpath* and *Cwd::extLibpath_set*.

DLL forwarder generation

Assume that the old DLL is named *perlE0AC.dll* (as is one for 5.005_53), and the new version is 5.6.1. Create a file *perl5shim.def-leader* with

```
  LIBRARY 'perlE0AC' INITINSTANCE TERMINSTANCE
  DESCRIPTION '@#perl5-porters@perl.org:5.006001#@ Perl module for 5.00553
-> Perl 5.6.1 forwarder'
  CODE LOADONCALL
  DATA LOADONCALL NONSHARED MULTIPLE
  EXPORTS
```

modifying the versions/names as needed. Run

```
 perl -wnle "next if 0../EXPORTS/; print qq(  \"$1\") if /\"(\w+)\"/"
perl5.def >lst
```

in the Perl build directory (to make the DLL smaller replace perl5.def with the definition file for the older version of Perl if present).

```
 cat perl5shim.def-leader lst >perl5shim.def
 gcc -Zomf -Zdll -o perlE0AC.dll perl5shim.def -s -llibperl
```

(ignore multiple `warning L4085`).

Threading

As of release 5.003_01 perl is linked to multithreaded C RTL DLL. If perl itself is not compiled multithread-enabled, so will not be perl's malloc(). However, extensions may use multiple thread on their own risk.

This was needed to compile `Perl/Tk` for XFree86-OS/2 out-of-the-box, and link with DLLs for other useful libraries, which typically are compiled with `-Zmt -Zcrtdll`.

Calls to external programs

Due to a popular demand the perl external program calling has been changed wrt Andreas Kaiser's port. *If* perl needs to call an external program *via shell*, the *f:/bin/sh.exe* will be called, or whatever is the override, see *PERL_SH_DIR*.

Thus means that you need to get some copy of a *sh.exe* as well (I use one from pdksh). The path *F:/bin* above is set up automatically during the build to a correct value on the builder machine, but is overridable at runtime,

Reasons: a consensus on `perl5-porters` was that perl should use one non-overridable shell per platform. The obvious choices for OS/2 are *cmd.exe* and *sh.exe*. Having perl build itself would be impossible with *cmd.exe* as a shell, thus I picked up `sh.exe`. This assures almost 100% compatibility with the scripts coming from *nix. As an added benefit this works as well under DOS if you use DOS-enabled port of pdksh (see *Prerequisites*).

Disadvantages: currently *sh.exe* of pdksh calls external programs via fork()/exec(), and there is *no* functioning exec() on OS/2. exec() is emulated by EMX by an asynchronous call while the caller waits for child completion (to pretend that the `pid` did not change). This means that 1 *extra* copy of *sh.exe* is made active via fork()/exec(), which may lead to some resources taken from the system (even if we do not count extra work needed for fork()ing).

Note that this a lesser issue now when we do not spawn *sh.exe* unless needed (metachars found).

One can always start *cmd.exe* explicitly via

```
system 'cmd', '/c', 'mycmd', 'arg1', 'arg2', ...
```

If you need to use *cmd.exe*, and do not want to hand-edit thousands of your scripts, the long-term solution proposed on p5-p is to have a directive

```
use OS2::Cmd;
```

which will override system(), exec(), ``, and `open(,'...|')`. With current perl you may override only system(), readpipe() - the explicit version of ``, and maybe exec(). The code will substitute the one-argument call to system() by `CORE::system('cmd.exe', '/c', shift)`.

If you have some working code for `OS2::Cmd`, please send it to me, I will include it into distribution. I have no need for such a module, so cannot test it.

For the details of the current situation with calling external programs, see *"2 (and DOS) programs under Perl" in Starting OS*. Set us mention a couple of features:

- External scripts may be called by their basename. Perl will try the same extensions as when processing **-S** command-line switch.
- External scripts starting with `#!` or `extproc` will be executed directly, without calling the shell, by calling the program specified on the rest of the first line.

Memory allocation

Perl uses its own malloc() under OS/2 - interpreters are usually malloc-bound for speed, but perl is not, since its malloc is lightning-fast. Perl-memory-usage-tuned benchmarks show that Perl's malloc is 5 times quicker than EMX one. I do not have convincing data about memory footprint, but a (pretty random) benchmark showed that Perl's one is 5% better.

Combination of perl's malloc() and rigid DLL name resolution creates a special problem with library functions which expect their return value to be free()d by system's free(). To facilitate extensions

which need to call such functions, system memory-allocation functions are still available with the prefix `emx_` added. (Currently only DLL perl has this, it should propagate to *perl_.exe* shortly.)

Threads

One can build perl with thread support enabled by providing `-D usethreads` option to *Configure*. Currently OS/2 support of threads is very preliminary.

Most notable problems:

`COND_WAIT`

may have a race condition (but probably does not due to edge-triggered nature of OS/2 Event semaphores). (Needs a reimplementation (in terms of chaining waiting threads, with the linked list stored in per-thread structure?)?)

os2.c

has a couple of static variables used in OS/2-specific functions. (Need to be moved to per-thread structure, or serialized?)

Note that these problems should not discourage experimenting, since they have a low probability of affecting small programs.

BUGS

This description is not updated often (since 5.6.1?), see *./os2/Changes* (*perlos2delta*) for more info.

AUTHOR

Ilya Zakharevich, cpan@ilyaz.org

SEE ALSO

perl(1).