NAME

perlport - Writing portable Perl

DESCRIPTION

Perl runs on numerous operating systems. While most of them share much in common, they also have their own unique features.

This document is meant to help you to find out what constitutes portable Perl code. That way once you make a decision to write portably, you know where the lines are drawn, and you can stay within them.

There is a tradeoff between taking full advantage of one particular type of computer and taking advantage of a full range of them. Naturally, as you broaden your range and become more diverse, the common factors drop, and you are left with an increasingly smaller area of common ground in which you can operate to accomplish a particular task. Thus, when you begin attacking a problem, it is important to consider under which part of the tradeoff curve you want to operate. Specifically, you must decide whether it is important that the task that you are coding have the full generality of being portable, or whether to just get the job done right now. This is the hardest choice to be made. The rest is easy, because Perl provides many choices, whichever way you want to approach your problem.

Looking at it another way, writing portable code is usually about willfully limiting your available choices. Naturally, it takes discipline and sacrifice to do that. The product of portability and convenience may be a constant. You have been warned.

Be aware of two important points:

Not all Perl programs have to be portable

> There is no reason you should not use Perl as a language to glue Unix tools together, or to prototype a Macintosh application, or to manage the Windows registry. If it makes no sense to aim for portability for one reason or another in a given program, then don't bother.

Nearly all of Perl already *is* portable

> Don't be fooled into thinking that it is hard to create portable Perl code. It isn't. Perl tries its level-best to bridge the gaps between what's available on different platforms, and all the means available to use those features. Thus almost all Perl code runs on any machine without modification. But there are some significant issues in writing portable code, and this document is entirely about those issues.

Here's the general rule: When you approach a task commonly done using a whole range of platforms, think about writing portable code. That way, you don't sacrifice much by way of the implementation choices you can avail yourself of, and at the same time you can give your users lots of platform choices. On the other hand, when you have to take advantage of some unique feature of a particular platform, as is often the case with systems programming (whether for Unix, Windows, VMS, etc.), consider writing platform-specific code.

When the code will run on only two or three operating systems, you may need to consider only the differences of those particular systems. The important thing is to decide where the code will run and to be deliberate in your decision.

The material below is separated into three main sections: main issues of portability (*ISSUES*), platform-specific issues (*PLATFORMS*), and built-in perl functions that behave differently on various ports (*FUNCTION IMPLEMENTATIONS*).

This information should not be considered complete; it includes possibly transient information about idiosyncrasies of some of the ports, almost all of which are in a state of constant evolution. Thus, this material should be considered a perpetual work in progress (<IMG SRC="yellow_sign.gif" ALT="Under Construction">).

ISSUES

Newlines

In most operating systems, lines in files are terminated by newlines. Just what is used as a newline may vary from OS to OS. Unix traditionally uses `\012`, one type of DOSish I/O uses `\015\012`, and Mac OS uses `\015`.

Perl uses `\n` to represent the "logical" newline, where what is logical may depend on the platform in use. In MacPerl, `\n` always means `\015`. In DOSish perls, `\n` usually means `\012`, but when accessing a file in "text" mode, perl uses the `:crlf` layer that translates it to (or from) `\015\012`, depending on whether you're reading or writing. Unix does the same thing on ttys in canonical mode. `\015\012` is commonly referred to as CRLF.

To trim trailing newlines from text lines use chomp(). With default settings that function looks for a trailing `\n` character and thus trims in a portable way.

When dealing with binary files (or text files in binary mode) be sure to explicitly set $/ to the appropriate value for your file format before using chomp().

Because of the "text" mode translation, DOSish perls have limitations in using `seek` and `tell` on a file accessed in "text" mode. Stick to `seek`-ing to locations you got from `tell` (and no others), and you are usually free to use `seek` and `tell` even in "text" mode. Using `seek` or `tell` or other file operations may be non-portable. If you use `binmode` on a file, however, you can usually `seek` and `tell` with arbitrary values in safety.

A common misconception in socket programming is that `\n` eq `\012` everywhere. When using protocols such as common Internet protocols, `\012` and `\015` are called for specifically, and the values of the logical `\n` and `\r` (carriage return) are not reliable.

```
print SOCKET "Hi there, client!\r\n";      # WRONG
print SOCKET "Hi there, client!\015\012";  # RIGHT
```

However, using `\015\012` (or `\cM\cJ`, or `\x0D\x0A`) can be tedious and unsightly, as well as confusing to those maintaining the code. As such, the Socket module supplies the Right Thing for those who want it.

```
use Socket qw(:DEFAULT :crlf);
print SOCKET "Hi there, client!$CRLF"      # RIGHT
```

When reading from a socket, remember that the default input record separator `$/` is `\n`, but robust socket code will recognize as either `\012` or `\015\012` as end of line:

```
while (<SOCKET>) {
    # ...
}
```

Because both CRLF and LF end in LF, the input record separator can be set to LF and any CR stripped later. Better to write:

```
use Socket qw(:DEFAULT :crlf);
local($/) = LF;      # not needed if $/ is already \012

while (<SOCKET>) {
    s/$CR?$LF/\n/;   # not sure if socket uses LF or CRLF, OK
#   s/\015?\012/\n/; # same thing
}
```

This example is preferred over the previous one--even for Unix platforms--because now any `\015`'s (

\cM's) are stripped out (and there was much rejoicing).

Similarly, functions that return text data--such as a function that fetches a web page--should sometimes translate newlines before returning the data, if they've not yet been translated to the local newline representation. A single line of code will often suffice:

```
$data =~ s/\015?\012/\n/g;
return $data;
```

Some of this may be confusing. Here's a handy reference to the ASCII CR and LF characters. You can print it out and stick it in your wallet.

```
LF  eq  \012  eq  \x0A  eq  \cJ  eq  chr(10)  eq  ASCII 10
CR  eq  \015  eq  \x0D  eq  \cM  eq  chr(13)  eq  ASCII 13
```

```
     | Unix | DOS  | Mac  |
---------------------------
\n   |  LF  |  LF  |  CR  |
\r   |  CR  |  CR  |  LF  |
\n * |  LF  | CRLF |  CR  |
\r * |  CR  |  CR  |  LF  |
---------------------------
* text-mode STDIO
```

The Unix column assumes that you are not accessing a serial line (like a tty) in canonical mode. If you are, then CR on input becomes "\n", and "\n" on output becomes CRLF.

These are just the most common definitions of \n and \r in Perl. There may well be others. For example, on an EBCDIC implementation such as z/OS (OS/390) or OS/400 (using the ILE, the PASE is ASCII-based) the above material is similar to "Unix" but the code numbers change:

```
LF  eq  \025  eq  \x15  eq  \cU  eq  chr(21)  eq  CP-1047 21
LF  eq  \045  eq  \x25  eq        chr(37)  eq  CP-0037 37
CR  eq  \015  eq  \x0D  eq  \cM  eq  chr(13)  eq  CP-1047 13
CR  eq  \015  eq  \x0D  eq  \cM  eq  chr(13)  eq  CP-0037 13
```

```
     | z/OS | OS/400 |
----------------------
\n   |  LF  |  LF    |
\r   |  CR  |  CR    |
\n * |  LF  |  LF    |
\r * |  CR  |  CR    |
----------------------
* text-mode STDIO
```

Numbers endianness and Width

Different CPUs store integers and floating point numbers in different orders (called *endianness*) and widths (32-bit and 64-bit being the most common today). This affects your programs when they attempt to transfer numbers in binary format from one CPU architecture to another, usually either "live" via network connection, or by storing the numbers to secondary storage such as a disk file or tape.

Conflicting storage orders make utter mess out of the numbers. If a little-endian host (Intel, VAX) stores 0x12345678 (305419896 in decimal), a big-endian host (Motorola, Sparc, PA) reads it as 0x78563412 (2018915346 in decimal). Alpha and MIPS can be either: Digital/Compaq used/uses them in little-endian mode; SGI/Cray uses them in big-endian mode. To avoid this problem in network

(socket) connections use the `pack` and `unpack` formats `n` and `N`, the "network" orders. These are guaranteed to be portable.

As of perl 5.9.2, you can also use the > and < modifiers to force big- or little-endian byte-order. This is useful if you want to store signed integers or 64-bit integers, for example.

You can explore the endianness of your platform by unpacking a data structure packed in native format such as:

```
print unpack("h*", pack("s2", 1, 2)), "\n";
# '10002000' on e.g. Intel x86 or Alpha 21064 in little-endian mode
# '00100020' on e.g. Motorola 68040
```

If you need to distinguish between endian architectures you could use either of the variables set like so:

```
$is_big_endian   = unpack("h*", pack("s", 1)) =~ /01/;
$is_little_endian = unpack("h*", pack("s", 1)) =~ /^1/;
```

Differing widths can cause truncation even between platforms of equal endianness. The platform of shorter width loses the upper parts of the number. There is no good solution for this problem except to avoid transferring or storing raw binary numbers.

One can circumnavigate both these problems in two ways. Either transfer and store numbers always in text format, instead of raw binary, or else consider using modules like Data::Dumper (included in the standard distribution as of Perl 5.005) and Storable (included as of perl 5.8). Keeping all data as text significantly simplifies matters.

The v-strings are portable only up to v2147483647 (0x7FFFFFFF), that's how far EBCDIC, or more precisely UTF-EBCDIC will go.

Files and Filesystems

Most platforms these days structure files in a hierarchical fashion. So, it is reasonably safe to assume that all platforms support the notion of a "path" to uniquely identify a file on the system. How that path is really written, though, differs considerably.

Although similar, file path specifications differ between Unix, Windows, Mac OS, OS/2, VMS, VOS, RISC OS, and probably others. Unix, for example, is one of the few OSes that has the elegant idea of a single root directory.

DOS, OS/2, VMS, VOS, and Windows can work similarly to Unix with `/` as path separator, or in their own idiosyncratic ways (such as having several root directories and various "unrooted" device files such NIL: and LPT:).

Mac OS 9 and earlier used `:` as a path separator instead of `/`.

The filesystem may support neither hard links (`link`) nor symbolic links (`symlink`, `readlink`, `lstat`).

The filesystem may support neither access timestamp nor change timestamp (meaning that about the only portable timestamp is the modification timestamp), or one second granularity of any timestamps (e.g. the FAT filesystem limits the time granularity to two seconds).

The "inode change timestamp" (the `-C` filetest) may really be the "creation timestamp" (which it is not in Unix).

VOS perl can emulate Unix filenames with `/` as path separator. The native pathname characters greater-than, less-than, number-sign, and percent-sign are always accepted.

RISC OS perl can emulate Unix filenames with `/` as path separator, or go native and use `.` for path

separator and `:` to signal filesystems and disk names.

Don't assume Unix filesystem access semantics: that read, write, and execute are all the permissions there are, and even if they exist, that their semantics (for example what do r, w, and x mean on a directory) are the Unix ones. The various Unix/POSIX compatibility layers usually try to make interfaces like chmod() work, but sometimes there simply is no good mapping.

If all this is intimidating, have no (well, maybe only a little) fear. There are modules that can help. The File::Spec modules provide methods to do the Right Thing on whatever platform happens to be running the program.

```
use File::Spec::Functions;
chdir(updir());        # go up one directory
my $file = catfile(curdir(), 'temp', 'file.txt');
# on Unix and Win32, './temp/file.txt'
# on Mac OS Classic, ':temp:file.txt'
# on VMS, '[.temp]file.txt'
```

File::Spec is available in the standard distribution as of version 5.004_05. File::Spec::Functions is only in File::Spec 0.7 and later, and some versions of perl come with version 0.6. If File::Spec is not updated to 0.7 or later, you must use the object-oriented interface from File::Spec (or upgrade File::Spec).

In general, production code should not have file paths hardcoded. Making them user-supplied or read from a configuration file is better, keeping in mind that file path syntax varies on different machines.

This is especially noticeable in scripts like Makefiles and test suites, which often assume `/` as a path separator for subdirectories.

Also of use is File::Basename from the standard distribution, which splits a pathname into pieces (base filename, full path to directory, and file suffix).

Even when on a single platform (if you can call Unix a single platform), remember not to count on the existence or the contents of particular system-specific files or directories, like */etc/passwd*, */etc/sendmail.conf*, */etc/resolv.conf*, or even */tmp/*. For example, */etc/passwd* may exist but not contain the encrypted passwords, because the system is using some form of enhanced security. Or it may not contain all the accounts, because the system is using NIS. If code does need to rely on such a file, include a description of the file and its format in the code's documentation, then make it easy for the user to override the default location of the file.

Don't assume a text file will end with a newline. They should, but people forget.

Do not have two files or directories of the same name with different case, like *test.pl* and *Test.pl*, as many platforms have case-insensitive (or at least case-forgiving) filenames. Also, try not to have non-word characters (except for `.`) in the names, and keep them to the 8.3 convention, for maximum portability, onerous a burden though this may appear.

Likewise, when using the AutoSplit module, try to keep your functions to 8.3 naming and case-insensitive conventions; or, at the least, make it so the resulting files have a unique (case-insensitively) first 8 characters.

Whitespace in filenames is tolerated on most systems, but not all, and even on systems where it might be tolerated, some utilities might become confused by such whitespace.

Many systems (DOS, VMS ODS-2) cannot have more than one `.` in their filenames.

Don't assume > won't be the first character of a filename. Always use < explicitly to open a file for reading, or even better, use the three-arg version of open, unless you want the user to be able to specify a pipe open.

```
    open my $fh, '<', $existing_file) or die $!;
```

If filenames might use strange characters, it is safest to open it with `sysopen` instead of `open`. `open` is magic and can translate characters like >, <, and `|`, which may be the wrong thing to do. (Sometimes, though, it's the right thing.) Three-arg open can also help protect against this translation in cases where it is undesirable.

Don't use `:` as a part of a filename since many systems use that for their own semantics (Mac OS Classic for separating pathname components, many networking schemes and utilities for separating the nodename and the pathname, and so on). For the same reasons, avoid `@`, `;` and `|`.

Don't assume that in pathnames you can collapse two leading slashes `//` into one: some networking and clustering filesystems have special semantics for that. Let the operating system to sort it out.

The *portable filename characters* as defined by ANSI C are

```
a b c d e f g h i j k l m n o p q r t u v w x y z
A B C D E F G H I J K L M N O P Q R T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
. _ -
```

and the "-" shouldn't be the first character. If you want to be hypercorrect, stay case-insensitive and within the 8.3 naming convention (all the files and directories have to be unique within one directory if their names are lowercased and truncated to eight characters before the `.`, if any, and to three characters after the `.`, if any). (And do not use `.s` in directory names.)

System Interaction

Not all platforms provide a command line. These are usually platforms that rely primarily on a Graphical User Interface (GUI) for user interaction. A program requiring a command line interface might not work everywhere. This is probably for the user of the program to deal with, so don't stay up late worrying about it.

Some platforms can't delete or rename files held open by the system, this limitation may also apply to changing filesystem metainformation like file permissions or owners. Remember to `close` files when you are done with them. Don't `unlink` or `rename` an open file. Don't `tie` or `open` a file already tied or opened; `untie` or `close` it first.

Don't open the same file more than once at a time for writing, as some operating systems put mandatory locks on such files.

Don't assume that write/modify permission on a directory gives the right to add or delete files/directories in that directory. That is filesystem specific: in some filesystems you need write/modify permission also (or even just) in the file/directory itself. In some filesystems (AFS, DFS) the permission to add/delete directory entries is a completely separate permission.

Don't assume that a single `unlink` completely gets rid of the file: some filesystems (most notably the ones in VMS) have versioned filesystems, and unlink() removes only the most recent one (it doesn't remove all the versions because by default the native tools on those platforms remove just the most recent version, too). The portable idiom to remove all the versions of a file is

```
    1 while unlink "file";
```

This will terminate if the file is undeleteable for some reason (protected, not there, and so on).

Don't count on a specific environment variable existing in `%ENV`. Don't count on `%ENV` entries being case-sensitive, or even case-preserving. Don't try to clear %ENV by saying `%ENV = ();`, or, if you really have to, make it conditional on `$^O ne 'VMS'` since in VMS the `%ENV` table is much more than a per-process key-value string table.

On VMS, some entries in the %ENV hash are dynamically created when their key is used on a read if they did not previously exist. The values for `$ENV{HOME}`, `$ENV{TERM}`, `$ENV{HOME}`, and `$ENV{USER}`, are known to be dynamically generated. The specific names that are dynamically generated may vary with the version of the C library on VMS, and more may exist than is documented.

On VMS by default, changes to the %ENV hash are persistent after the process exits. This can cause unintended issues.

Don't count on signals or `%SIG` for anything.

Don't count on filename globbing. Use `opendir`, `readdir`, and `closedir` instead.

Don't count on per-program environment variables, or per-program current directories.

Don't count on specific values of `$!`, neither numeric nor especially the strings values. Users may switch their locales causing error messages to be translated into their languages. If you can trust a POSIXish environment, you can portably use the symbols defined by the Errno module, like ENOENT. And don't trust on the values of `$!` at all except immediately after a failed system call.

Command names versus file pathnames

Don't assume that the name used to invoke a command or program with `system` or `exec` can also be used to test for the existence of the file that holds the executable code for that command or program. First, many systems have "internal" commands that are built-in to the shell or OS and while these commands can be invoked, there is no corresponding file. Second, some operating systems (e.g., Cygwin, DJGPP, OS/2, and VOS) have required suffixes for executable files; these suffixes are generally permitted on the command name but are not required. Thus, a command like "perl" might exist in a file named "perl", "perl.exe", or "perl.pm", depending on the operating system. The variable "_exe" in the Config module holds the executable suffix, if any. Third, the VMS port carefully sets up $^X and $Config{perlpath} so that no further processing is required. This is just as well, because the matching regular expression used below would then have to deal with a possible trailing version number in the VMS file name.

To convert $^X to a file pathname, taking account of the requirements of the various operating system possibilities, say:

```
use Config;
my $thisperl = $^X;
if ($^O ne 'VMS')
   {$thisperl .= $Config{_exe} unless $thisperl =~ m/$Config{_exe}$/i;}
```

To convert $Config{perlpath} to a file pathname, say:

```
use Config;
my $thisperl = $Config{perlpath};
if ($^O ne 'VMS')
   {$thisperl .= $Config{_exe} unless $thisperl =~ m/$Config{_exe}$/i;}
```

Networking

Don't assume that you can reach the public Internet.

Don't assume that there is only one way to get through firewalls to the public Internet.

Don't assume that you can reach outside world through any other port than 80, or some web proxy. ftp is blocked by many firewalls.

Don't assume that you can send email by connecting to the local SMTP port.

Don't assume that you can reach yourself or any node by the name 'localhost'. The same goes for

'127.0.0.1'. You will have to try both.

Don't assume that the host has only one network card, or that it can't bind to many virtual IP addresses.

Don't assume a particular network device name.

Don't assume a particular set of ioctl()s will work.

Don't assume that you can ping hosts and get replies.

Don't assume that any particular port (service) will respond.

Don't assume that Sys::Hostname (or any other API or command) returns either a fully qualified hostname or a non-qualified hostname: it all depends on how the system had been configured. Also remember that for things such as DHCP and NAT, the hostname you get back might not be very useful.

All the above "don't":s may look daunting, and they are, but the key is to degrade gracefully if one cannot reach the particular network service one wants. Croaking or hanging do not look very professional.

Interprocess Communication (IPC)

In general, don't directly access the system in code meant to be portable. That means, no `system`, `exec`, `fork`, `pipe`, ``` `` ```, `qx//`, `open` with a `|`, nor any of the other things that makes being a perl hacker worth being.

Commands that launch external processes are generally supported on most platforms (though many of them do not support any type of forking). The problem with using them arises from what you invoke them on. External tools are often named differently on different platforms, may not be available in the same location, might accept different arguments, can behave differently, and often present their results in a platform-dependent way. Thus, you should seldom depend on them to produce consistent results. (Then again, if you're calling *netstat -a*, you probably don't expect it to run on both Unix and CP/M.)

One especially common bit of Perl code is opening a pipe to **sendmail**:

```
    open(MAIL, '|/usr/lib/sendmail -t')
 or die "cannot fork sendmail: $!";
```

This is fine for systems programming when sendmail is known to be available. But it is not fine for many non-Unix systems, and even some Unix systems that may not have sendmail installed. If a portable solution is needed, see the various distributions on CPAN that deal with it. Mail::Mailer and Mail::Send in the MailTools distribution are commonly used, and provide several mailing methods, including mail, sendmail, and direct SMTP (via Net::SMTP) if a mail transfer agent is not available. Mail::Sendmail is a standalone module that provides simple, platform-independent mailing.

The Unix System V IPC (`msg*(), sem*(), shm*()`) is not available even on all Unix platforms.

Do not use either the bare result of `pack("N", 10, 20, 30, 40)` or bare v-strings (such as `v10.20.30.40`) to represent IPv4 addresses: both forms just pack the four bytes into network order. That this would be equal to the C language `in_addr` struct (which is what the socket code internally uses) is not guaranteed. To be portable use the routines of the Socket extension, such as `inet_aton()`, `inet_ntoa()`, and `sockaddr_in()`.

The rule of thumb for portable code is: Do it all in portable Perl, or use a module (that may internally implement it with platform-specific code, but expose a common interface).

External Subroutines (XS)

XS code can usually be made to work with any platform, but dependent libraries, header files, etc., might not be readily available or portable, or the XS code itself might be platform-specific, just as Perl code might be. If the libraries and headers are portable, then it is normally reasonable to make sure the XS code is portable, too.

A different type of portability issue arises when writing XS code: availability of a C compiler on the end-user's system. C brings with it its own portability issues, and writing XS code will expose you to some of those. Writing purely in Perl is an easier way to achieve portability.

Standard Modules

In general, the standard modules work across platforms. Notable exceptions are the CPAN module (which currently makes connections to external programs that may not be available), platform-specific modules (like ExtUtils::MM_VMS), and DBM modules.

There is no one DBM module available on all platforms. SDBM_File and the others are generally available on all Unix and DOSish ports, but not in MacPerl, where only NBDM_File and DB_File are available.

The good news is that at least some DBM module should be available, and AnyDBM_File will use whichever module it can find. Of course, then the code needs to be fairly strict, dropping to the greatest common factor (e.g., not exceeding 1K for each record), so that it will work with any DBM module. See *AnyDBM_File* for more details.

Time and Date

The system's notion of time of day and calendar date is controlled in widely different ways. Don't assume the timezone is stored in `$ENV{TZ}`, and even if it is, don't assume that you can control the timezone through that variable. Don't assume anything about the three-letter timezone abbreviations (for example that MST would be the Mountain Standard Time, it's been known to stand for Moscow Standard Time). If you need to use timezones, express them in some unambiguous format like the exact number of minutes offset from UTC, or the POSIX timezone format.

Don't assume that the epoch starts at 00:00:00, January 1, 1970, because that is OS- and implementation-specific. It is better to store a date in an unambiguous representation. The ISO 8601 standard defines YYYY-MM-DD as the date format, or YYYY-MM-DDTHH-MM-SS (that's a literal "T" separating the date from the time). Please do use the ISO 8601 instead of making us guess what date 02/03/04 might be. ISO 8601 even sorts nicely as-is. A text representation (like "1987-12-18") can be easily converted into an OS-specific value using a module like Date::Parse. An array of values, such as those returned by `localtime`, can be converted to an OS-specific representation using Time::Local.

When calculating specific times, such as for tests in time or date modules, it may be appropriate to calculate an offset for the epoch.

```
require Time::Local;
my $offset = Time::Local::timegm(0, 0, 0, 1, 0, 70);
```

The value for `$offset` in Unix will be `0`, but in Mac OS Classic will be some large number. `$offset` can then be added to a Unix time value to get what should be the proper value on any system.

Character sets and character encoding

Assume very little about character sets.

Assume nothing about numerical values (`ord`, `chr`) of characters. Do not use explicit code point ranges (like \xHH-\xHH); use for example symbolic character classes like `[:print:]`.

Do not assume that the alphabetic characters are encoded contiguously (in the numeric sense). There may be gaps.

Do not assume anything about the ordering of the characters. The lowercase letters may come before or after the uppercase letters; the lowercase and uppercase may be interlaced so that both "a" and "A" come before "b"; the accented and other international characters may be interlaced so that ä comes before "b".

Internationalisation

If you may assume POSIX (a rather large assumption), you may read more about the POSIX locale system from *perllocale*. The locale system at least attempts to make things a little bit more portable, or at least more convenient and native-friendly for non-English users. The system affects character sets and encoding, and date and time formatting--amongst other things.

If you really want to be international, you should consider Unicode. See *perluniintro* and *perlunicode* for more information.

If you want to use non-ASCII bytes (outside the bytes 0x00..0x7f) in the "source code" of your code, to be portable you have to be explicit about what bytes they are. Someone might for example be using your code under a UTF-8 locale, in which case random native bytes might be illegal ("Malformed UTF-8 ...") This means that for example embedding ISO 8859-1 bytes beyond 0x7f into your strings might cause trouble later. If the bytes are native 8-bit bytes, you can use the `bytes` pragma. If the bytes are in a string (regular expression being a curious string), you can often also use the `\xHH` notation instead of embedding the bytes as-is. (If you want to write your code in UTF-8, you can use the `utf8`.) The `bytes` and `utf8` pragmata are available since Perl 5.6.0.

System Resources

If your code is destined for systems with severely constrained (or missing!) virtual memory systems then you want to be *especially* mindful of avoiding wasteful constructs such as:

```
# NOTE: this is no longer "bad" in perl5.005
for (0..10000000) {}                        # bad
for (my $x = 0; $x <= 10000000; ++$x) {}    # good

my @lines = <$very_large_file>;             # bad

while (<$fh>) {$file .= $_}                 # sometimes bad
my $file = join('', <$fh>);                 # better
```

The last two constructs may appear unintuitive to most people. The first repeatedly grows a string, whereas the second allocates a large chunk of memory in one go. On some systems, the second is more efficient that the first.

Security

Most multi-user platforms provide basic levels of security, usually implemented at the filesystem level. Some, however, unfortunately do not. Thus the notion of user id, or "home" directory, or even the state of being logged-in, may be unrecognizable on many platforms. If you write programs that are security-conscious, it is usually best to know what type of system you will be running under so that you can write code explicitly for that platform (or class of platforms).

Don't assume the Unix filesystem access semantics: the operating system or the filesystem may be using some ACL systems, which are richer languages than the usual rwx. Even if the rwx exist, their semantics might be different.

(From security viewpoint testing for permissions before attempting to do something is silly anyway: if one tries this, there is potential for race conditions. Someone or something might change the permissions between the permissions check and the actual operation. Just try the operation.)

Don't assume the Unix user and group semantics: especially, don't expect the $< and $> (or the $(and $)) to work for switching identities (or memberships).

Don't assume set-uid and set-gid semantics. (And even if you do, think twice: set-uid and set-gid are a known can of security worms.)

Style

For those times when it is necessary to have platform-specific code, consider keeping the platform-specific code in one place, making porting to other platforms easier. Use the Config module and the special variable $^O to differentiate platforms, as described in *PLATFORMS*.

Be careful in the tests you supply with your module or programs. Module code may be fully portable, but its tests might not be. This often happens when tests spawn off other processes or call external programs to aid in the testing, or when (as noted above) the tests assume certain things about the filesystem and paths. Be careful not to depend on a specific output style for errors, such as when checking `$!` after a failed system call. Using `$!` for anything else than displaying it as output is doubtful (though see the Errno module for testing reasonably portably for error value). Some platforms expect a certain output format, and Perl on those platforms may have been adjusted accordingly. Most specifically, don't anchor a regex when testing an error value.

CPAN Testers

Modules uploaded to CPAN are tested by a variety of volunteers on different platforms. These CPAN testers are notified by mail of each new upload, and reply to the list with PASS, FAIL, NA (not applicable to this platform), or UNKNOWN (unknown), along with any relevant notations.

The purpose of the testing is twofold: one, to help developers fix any problems in their code that crop up because of lack of testing on other platforms; two, to provide users with information about whether a given module works on a given platform.

Also see:

- Mailing list: cpan-testers-discuss@perl.org
- Testing results: *http://www.cpantesters.org/*

PLATFORMS

As of version 5.002, Perl is built with a $^O variable that indicates the operating system it was built on. This was implemented to help speed up code that would otherwise have to `use Config` and use the value of `$Config{osname}`. Of course, to get more detailed information about the system, looking into `%Config` is certainly recommended.

`%Config` cannot always be trusted, however, because it was built at compile time. If perl was built in one place, then transferred elsewhere, some values may be wrong. The values may even have been edited after the fact.

Unix

Perl works on a bewildering variety of Unix and Unix-like platforms (see e.g. most of the files in the *hints/* directory in the source code kit). On most of these systems, the value of $^O (hence `$Config{'osname'}`, too) is determined either by lowercasing and stripping punctuation from the first field of the string returned by typing `uname -a` (or a similar command) at the shell prompt or by testing the file system for the presence of uniquely named files such as a kernel or header file. Here, for example, are a few of the more popular Unix flavors:

```
uname         $^O        $Config{'archname'}
--------------------------------------------
AIX           aix        aix
BSD/OS        bsdos      i386-bsdos
Darwin        darwin     darwin
dgux          dgux       AViiON-dgux
DYNIX/ptx     dynixptx   i386-dynixptx
FreeBSD       freebsd    freebsd-i386
```

```
Haiku          haiku      BePC-haiku
Linux          linux      arm-linux
Linux          linux      i386-linux
Linux          linux      i586-linux
Linux          linux      ppc-linux
HP-UX          hpux       PA-RISC1.1
IRIX           irix       irix
Mac OS X       darwin     darwin
NeXT 3         next       next-fat
NeXT 4         next       OPENSTEP-Mach
openbsd        openbsd    i386-openbsd
OSF1           dec_osf    alpha-dec_osf
reliantunix-n  svr4       RM400-svr4
SCO_SV         sco_sv     i386-sco_sv
SINIX-N        svr4       RM400-svr4
sn4609         unicos     CRAY_C90-unicos
sn6521         unicosmk   t3e-unicosmk
sn9617         unicos     CRAY_J90-unicos
SunOS          solaris    sun4-solaris
SunOS          solaris    i86pc-solaris
SunOS4         sunos      sun4-sunos
```

Because the value of `$Config{archname}` may depend on the hardware architecture, it can vary more than the value of `$^O`.

DOS and Derivatives

Perl has long been ported to Intel-style microcomputers running under systems like PC-DOS, MS-DOS, OS/2, and most Windows platforms you can bring yourself to mention (except for Windows CE, if you count that). Users familiar with *COMMAND.COM* or *CMD.EXE* style shells should be aware that each of these file specifications may have subtle differences:

```
my $filespec0 = "c:/foo/bar/file.txt";
my $filespec1 = "c:\\foo\\bar\\file.txt";
my $filespec2 = 'c:\foo\bar\file.txt';
my $filespec3 = 'c:\\foo\\bar\\file.txt';
```

System calls accept either `/` or `\` as the path separator. However, many command-line utilities of DOS vintage treat `/` as the option prefix, so may get confused by filenames containing `/`. Aside from calling any external programs, `/` will work just fine, and probably better, as it is more consistent with popular usage, and avoids the problem of remembering what to backwhack and what not to.

The DOS FAT filesystem can accommodate only "8.3" style filenames. Under the "case-insensitive, but case-preserving" HPFS (OS/2) and NTFS (NT) filesystems you may have to be careful about case returned with functions like `readdir` or used with functions like `open` or `opendir`.

DOS also treats several filenames as special, such as AUX, PRN, NUL, CON, COM1, LPT1, LPT2, etc. Unfortunately, sometimes these filenames won't even work if you include an explicit directory prefix. It is best to avoid such filenames, if you want your code to be portable to DOS and its derivatives. It's hard to know what these all are, unfortunately.

Users of these operating systems may also wish to make use of scripts such as *pl2bat.bat* or *pl2cmd* to put wrappers around your scripts.

Newline (`\n`) is translated as `\015\012` by STDIO when reading from and writing to files (see *Newlines*). `binmode(FILEHANDLE)` will keep `\n` translated as `\012` for that filehandle. Since it is a no-op on other systems, `binmode` should be used for cross-platform code that deals with binary data. That's assuming you realize in advance that your data is in binary. General-purpose programs should

often assume nothing about their data.

The $^O variable and the $Config{archname} values for various DOSish perls are as follows:

```
OS            $^O      $Config{archname}   ID    Version
--------------------------------------------------------
MS-DOS        dos        ?
PC-DOS        dos        ?
OS/2          os2        ?
Windows 3.1   ?          ?                 0      3 01
Windows 95    MSWin32    MSWin32-x86       1      4 00
Windows 98    MSWin32    MSWin32-x86       1      4 10
Windows ME    MSWin32    MSWin32-x86       1      ?
Windows NT    MSWin32    MSWin32-x86       2      4 xx
Windows NT    MSWin32    MSWin32-ALPHA     2      4 xx
Windows NT    MSWin32    MSWin32-ppc       2      4 xx
Windows 2000  MSWin32    MSWin32-x86       2      5 00
Windows XP    MSWin32    MSWin32-x86       2      5 01
Windows 2003  MSWin32    MSWin32-x86       2      5 02
Windows Vista MSWin32    MSWin32-x86       2      6 00
Windows 7     MSWin32    MSWin32-x86       2      6 01
Windows 7     MSWin32    MSWin32-x64       2      6 01
Windows CE    MSWin32    ?                 3
Cygwin        cygwin     cygwin
```

The various MSWin32 Perl's can distinguish the OS they are running on via the value of the fifth element of the list returned from Win32::GetOSVersion(). For example:

```
if ($^O eq 'MSWin32') {
    my @os_version_info = Win32::GetOSVersion();
    print +('3.1','95','NT')[$os_version_info[4]],"\n";
}
```

There are also Win32::IsWinNT() and Win32::IsWin95(), try `perldoc Win32`, and as of libwin32 0.19 (not part of the core Perl distribution) Win32::GetOSName(). The very portable POSIX::uname() will work too:

```
c:\> perl -MPOSIX -we "print join '|', uname"
Windows NT|moonru|5.0|Build 2195 (Service Pack 2)|x86
```

Also see:

- The djgpp environment for DOS, *http://www.delorie.com/djgpp/* and *perldos*.
- The EMX environment for DOS, OS/2, etc. emx@iaehv.nl, *ftp://hobbes.nmsu.edu/pub/os2/dev/emx/* Also *perlos2*.
- Build instructions for Win32 in *perlwin32*, or under the Cygnus environment in *perlcygwin*.
- The `Win32::*` modules in *Win32*.
- The ActiveState Pages, *http://www.activestate.com/*
- The Cygwin environment for Win32; *README.cygwin* (installed as *perlcygwin*), *http://www.cygwin.com/*
- The U/WIN environment for Win32, *http://www.research.att.com/sw/tools/uwin/*
- Build instructions for OS/2, *perlos2*

VMS

Perl on VMS is discussed in *perlvms* in the perl distribution.

The official name of VMS as of this writing is OpenVMS.

Perl on VMS can accept either VMS- or Unix-style file specifications as in either of the following:

```
$ perl -ne "print if /perl_setup/i" SYS$LOGIN:LOGIN.COM
$ perl -ne "print if /perl_setup/i" /sys$login/login.com
```

but not a mixture of both as in:

```
$ perl -ne "print if /perl_setup/i" sys$login:/login.com
Can't open sys$login:/login.com: file specification syntax error
```

Interacting with Perl from the Digital Command Language (DCL) shell often requires a different set of quotation marks than Unix shells do. For example:

```
$ perl -e "print ""Hello, world.\n"""
Hello, world.
```

There are several ways to wrap your perl scripts in DCL *.COM* files, if you are so inclined. For example:

```
$ write sys$output "Hello from DCL!"
$ if p1 .eqs. ""
$ then perl -x 'f$environment("PROCEDURE")
$ else perl -x - 'p1 'p2 'p3 'p4 'p5 'p6 'p7 'p8
$ deck/dollars="__END__"
#!/usr/bin/perl

print "Hello from Perl!\n";

__END__
$ endif
```

Do take care with `$ ASSIGN/nolog/user SYS$COMMAND: SYS$INPUT` if your perl-in-DCL script expects to do things like `$read = <STDIN>;`.

The VMS operating system has two filesystems, known as ODS-2 and ODS-5.

For ODS-2, filenames are in the format "name.extension;version". The maximum length for filenames is 39 characters, and the maximum length for extensions is also 39 characters. Version is a number from 1 to 32767. Valid characters are `/[A-Z0-9$_-]/`.

The ODS-2 filesystem is case-insensitive and does not preserve case. Perl simulates this by converting all filenames to lowercase internally.

For ODS-5, filenames may have almost any character in them and can include Unicode characters. Characters that could be misinterpreted by the DCL shell or file parsing utilities need to be prefixed with the ^ character, or replaced with hexadecimal characters prefixed with the ^ character. Such prefixing is only needed with the pathnames are in VMS format in applications. Programs that can accept the Unix format of pathnames do not need the escape characters. The maximum length for filenames is 255 characters. The ODS-5 file system can handle both a case preserved and a case sensitive mode.

ODS-5 is only available on the OpenVMS for 64 bit platforms.

Support for the extended file specifications is being done as optional settings to preserve backward compatibility with Perl scripts that assume the previous VMS limitations.

In general routines on VMS that get a Unix format file specification should return it in a Unix format, and when they get a VMS format specification they should return a VMS format unless they are documented to do a conversion.

For routines that generate return a file specification, VMS allows setting if the C library which Perl is built on if it will be returned in VMS format or in Unix format.

With the ODS-2 file system, there is not much difference in syntax of filenames without paths for VMS or Unix. With the extended character set available with ODS-5 there can be a significant difference.

Because of this, existing Perl scripts written for VMS were sometimes treating VMS and Unix filenames interchangeably. Without the extended character set enabled, this behavior will mostly be maintained for backwards compatibility.

When extended characters are enabled with ODS-5, the handling of Unix formatted file specifications is to that of a Unix system.

VMS file specifications without extensions have a trailing dot. An equivalent Unix file specification should not show the trailing dot.

The result of all of this, is that for VMS, for portable scripts, you can not depend on Perl to present the filenames in lowercase, to be case sensitive, and that the filenames could be returned in either Unix or VMS format.

And if a routine returns a file specification, unless it is intended to convert it, it should return it in the same format as it found it.

`readdir` by default has traditionally returned lowercased filenames. When the ODS-5 support is enabled, it will return the exact case of the filename on the disk.

Files without extensions have a trailing period on them, so doing a `readdir` in the default mode with a file named *A.;5* will return *a.* when VMS is (though that file could be opened with `open(FH, 'A')`).

With support for extended file specifications and if `opendir` was given a Unix format directory, a file named *A.;5* will return *a* and optionally in the exact case on the disk. When `opendir` is given a VMS format directory, then `readdir` should return *a.*, and again with the optionally the exact case.

RMS had an eight level limit on directory depths from any rooted logical (allowing 16 levels overall) prior to VMS 7.2, and even with versions of VMS on VAX up through 7.3. Hence `PERL_ROOT:[LIB.2.3.4.5.6.7.8]` is a valid directory specification but `PERL_ROOT:[LIB.2.3.4.5.6.7.8.9]` is not. *Makefile.PL* authors might have to take this into account, but at least they can refer to the former as `/PERL_ROOT/lib/2/3/4/5/6/7/8/`.

Pumpkings and module integrators can easily see whether files with too many directory levels have snuck into the core by running the following in the top-level source directory:

```
$ perl -ne "$_=~s/\s+.*//; print if scalar(split /\//) > 8;" < MANIFEST
```

The VMS::Filespec module, which gets installed as part of the build process on VMS, is a pure Perl module that can easily be installed on non-VMS platforms and can be helpful for conversions to and from RMS native formats. It is also now the only way that you should check to see if VMS is in a case sensitive mode.

What `\n` represents depends on the type of file opened. It usually represents `\012` but it could also be `\015`, `\012`, `\015\012`, `\000`, `\040`, or nothing depending on the file organization and record format. The VMS::Stdio module provides access to the special fopen() requirements of files with unusual attributes on VMS.

TCP/IP stacks are optional on VMS, so socket routines might not be implemented. UDP sockets may not be supported.

The TCP/IP library support for all current versions of VMS is dynamically loaded if present, so even if the routines are configured, they may return a status indicating that they are not implemented.

The value of `$^O` on OpenVMS is "VMS". To determine the architecture that you are running on without resorting to loading all of `%Config` you can examine the content of the `@INC` array like so:

```
if (grep(/VMS_AXP/, @INC)) {
    print "I'm on Alpha!\n";

} elsif (grep(/VMS_VAX/, @INC)) {
    print "I'm on VAX!\n";

} elsif (grep(/VMS_IA64/, @INC)) {
    print "I'm on IA64!\n";

} else {
    print "I'm not so sure about where $^O is...\n";
}
```

In general, the significant differences should only be if Perl is running on VMS_VAX or one of the 64 bit OpenVMS platforms.

On VMS, perl determines the UTC offset from the `SYS$TIMEZONE_DIFFERENTIAL` logical name. Although the VMS epoch began at 17-NOV-1858 00:00:00.00, calls to `localtime` are adjusted to count offsets from 01-JAN-1970 00:00:00.00, just like Unix.

Also see:

- *README.vms* (installed as *README_vms*), *perlvms*
- vmsperl list, vmsperl-subscribe@perl.org
- vmsperl on the web, *http://www.sidhe.org/vmsperl/index.html*

VOS

Perl on VOS (also known as OpenVOS) is discussed in *README.vos* in the perl distribution (installed as *perlvos*). Perl on VOS can accept either VOS- or Unix-style file specifications as in either of the following:

```
$ perl -ne "print if /perl_setup/i" >system>notices
$ perl -ne "print if /perl_setup/i" /system/notices
```

or even a mixture of both as in:

```
$ perl -ne "print if /perl_setup/i" >system/notices
```

Even though VOS allows the slash character to appear in object names, because the VOS port of Perl interprets it as a pathname delimiting character, VOS files, directories, or links whose names contain a slash character cannot be processed. Such files must be renamed before they can be processed by Perl.

Older releases of VOS (prior to OpenVOS Release 17.0) limit file names to 32 or fewer characters, prohibit file names from starting with a `-` character, and prohibit file names from containing any character matching `tr/ !#%&'()*;<=>?//`.

Newer releases of VOS (OpenVOS Release 17.0 or later) support a feature known as extended names. On these releases, file names can contain up to 255 characters, are prohibited from starting with a - character, and the set of prohibited characters is reduced to any character matching `tr/#%*<>?//`. There are restrictions involving spaces and apostrophes: these characters must not begin or end a name, nor can they immediately precede or follow a period. Additionally, a space must not immediately precede another space or hyphen. Specifically, the following character combinations are prohibited: space-space, space-hyphen, period-space, space-period, period-apostrophe, apostrophe-period, leading or trailing space, and leading or trailing apostrophe. Although an extended file name is limited to 255 characters, a path name is still limited to 256 characters.

The value of `$^O` on VOS is "VOS". To determine the architecture that you are running on without resorting to loading all of `%Config` you can examine the content of the @INC array like so:

```
if ($^O =~ /VOS/) {
    print "I'm on a Stratus box!\n";
} else {
    print "I'm not on a Stratus box!\n";
    die;
}
```

Also see:

- *README.vos* (installed as *perlvos*)
- The VOS mailing list.

  There is no specific mailing list for Perl on VOS. You can post comments to the comp.sys.stratus newsgroup, or use the contact information located in the distribution files on the Stratus Anonymous FTP site.
- VOS Perl on the web at *http://ftp.stratus.com/pub/vos/posix/posix.html*

EBCDIC Platforms

Recent versions of Perl have been ported to platforms such as OS/400 on AS/400 minicomputers as well as OS/390, VM/ESA, and BS2000 for S/390 Mainframes. Such computers use EBCDIC character sets internally (usually Character Code Set ID 0037 for OS/400 and either 1047 or POSIX-BC for S/390 systems). On the mainframe perl currently works under the "Unix system services for OS/390" (formerly known as OpenEdition), VM/ESA OpenEdition, or the BS200 POSIX-BC system (BS2000 is supported in perl 5.6 and greater). See *perlos390* for details. Note that for OS/400 there is also a port of Perl 5.8.1/5.9.0 or later to the PASE which is ASCII-based (as opposed to ILE which is EBCDIC-based), see *perlos400*.

As of R2.5 of USS for OS/390 and Version 2.3 of VM/ESA these Unix sub-systems do not support the `#!` shebang trick for script invocation. Hence, on OS/390 and VM/ESA perl scripts can be executed with a header similar to the following simple script:

```
: # use perl
    eval 'exec /usr/local/bin/perl -S $0 ${1+"$@"}'
        if 0;
#!/usr/local/bin/perl     # just a comment really

print "Hello from perl!\n";
```

OS/390 will support the `#!` shebang trick in release 2.8 and beyond. Calls to `system` and backticks can use POSIX shell syntax on all S/390 systems.

On the AS/400, if PERL5 is in your library list, you may need to wrap your perl scripts in a CL procedure to invoke them like so:

```
BEGIN
  CALL PGM(PERL5/PERL) PARM('/QOpenSys/hello.pl')
ENDPGM
```

This will invoke the perl script *hello.pl* in the root of the QOpenSys file system. On the AS/400 calls to `system` or backticks must use CL syntax.

On these platforms, bear in mind that the EBCDIC character set may have an effect on what happens with some perl functions (such as `chr`, `pack`, `print`, `printf`, `ord`, `sort`, `sprintf`, `unpack`), as well as bit-fiddling with ASCII constants using operators like `^`, `&` and `|`, not to mention dealing with socket interfaces to ASCII computers (see *Newlines*).

Fortunately, most web servers for the mainframe will correctly translate the `\n` in the following statement to its ASCII equivalent (`\r` is the same under both Unix and OS/390 & VM/ESA):

```
print "Content-type: text/html\r\n\r\n";
```

The values of `$^O` on some of these platforms includes:

```
uname         $^O        $Config{'archname'}
--------------------------------------------
OS/390        os390      os390
OS400         os400      os400
POSIX-BC      posix-bc   BS2000-posix-bc
VM/ESA        vmesa      vmesa
```

Some simple tricks for determining if you are running on an EBCDIC platform could include any of the following (perhaps all):

```
if ("\t" eq "\005")   { print "EBCDIC may be spoken here!\n"; }

if (ord('A') == 193) { print "EBCDIC may be spoken here!\n"; }

if (chr(169) eq 'z') { print "EBCDIC may be spoken here!\n"; }
```

One thing you may not want to rely on is the EBCDIC encoding of punctuation characters since these may differ from code page to code page (and once your module or script is rumoured to work with EBCDIC, folks will want it to work with all EBCDIC character sets).

Also see:

- *perlos390*, *README.os390*, *perlbs2000*, *README.vmesa*, *perlebcdic*.
- The perl-mvs@perl.org list is for discussion of porting issues as well as general usage issues for all EBCDIC Perls. Send a message body of "subscribe perl-mvs" to majordomo@perl.org.
- AS/400 Perl information at *http://as400.rochester.ibm.com/* as well as on CPAN in the *ports/* directory.

Acorn RISC OS

Because Acorns use ASCII with newlines (`\n`) in text files as `\012` like Unix, and because Unix filename emulation is turned on by default, most simple scripts will probably work "out of the box". The native filesystem is modular, and individual filesystems are free to be case-sensitive or insensitive, and are usually case-preserving. Some native filesystems have name length limits, which file and directory names are silently truncated to fit. Scripts should be aware that the standard filesystem currently has a name length limit of **10** characters, with up to 77 items in a directory, but other filesystems may not impose such limitations.

Native filenames are of the form

```
Filesystem#Special_Field::DiskName.$.Directory.Directory.File
```

where

```
Special_Field is not usually present, but may contain . and $ .
Filesystem =~ m|[A-Za-z0-9_]|
DsicName   =~ m|[A-Za-z0-9_/]|
$ represents the root directory
. is the path separator
@ is the current directory (per filesystem but machine global)
^ is the parent directory
Directory and File =~ m|[^\0- "\.\$\%\&:\@\\^\|\177]+|
```

The default filename translation is roughly `tr|/.|./|;`

Note that `"ADFS::HardDisk.$.File" ne 'ADFS::HardDisk.$.File'` and that the second stage of $ interpolation in regular expressions will fall foul of the `$.` if scripts are not careful.

Logical paths specified by system variables containing comma-separated search lists are also allowed; hence `System:Modules` is a valid filename, and the filesystem will prefix `Modules` with each section of `System$Path` until a name is made that points to an object on disk. Writing to a new file `System:Modules` would be allowed only if `System$Path` contains a single item list. The filesystem will also expand system variables in filenames if enclosed in angle brackets, so `<System$Dir>.Modules` would look for the file `$ENV{'System$Dir'} . 'Modules'`. The obvious implication of this is that **fully qualified filenames can start with <>** and should be protected when `open` is used for input.

Because `.` was in use as a directory separator and filenames could not be assumed to be unique after 10 characters, Acorn implemented the C compiler to strip the trailing `.c` `.h` `.s` and `.o` suffix from filenames specified in source code and store the respective files in subdirectories named after the suffix. Hence files are translated:

```
foo.h           h.foo
C:foo.h         C:h.foo         (logical path variable)
sys/os.h        sys.h.os        (C compiler groks Unix-speak)
10charname.c    c.10charname
10charname.o    o.10charname
11charname_.c   c.11charname    (assuming filesystem truncates at 10)
```

The Unix emulation library's translation of filenames to native assumes that this sort of translation is required, and it allows a user-defined list of known suffixes that it will transpose in this fashion. This may seem transparent, but consider that with these rules `foo/bar/baz.h` and `foo/bar/h/baz` both map to `foo.bar.h.baz`, and that `readdir` and `glob` cannot and do not attempt to emulate the reverse mapping. Other `.`'s in filenames are translated to `/`.

As implied above, the environment accessed through `%ENV` is global, and the convention is that program specific environment variables are of the form `Program$Name`. Each filesystem maintains a current directory, and the current filesystem's current directory is the **global** current directory. Consequently, sociable programs don't change the current directory but rely on full pathnames, and programs (and Makefiles) cannot assume that they can spawn a child process which can change the current directory without affecting its parent (and everyone else for that matter).

Because native operating system filehandles are global and are currently allocated down from 255, with 0 being a reserved value, the Unix emulation library emulates Unix filehandles. Consequently, you can't rely on passing `STDIN`, `STDOUT`, or `STDERR` to your children.

The desire of users to express filenames of the form `<Foo$Dir>.Bar` on the command line unquoted causes problems, too: `` ` `` command output capture has to perform a guessing game. It assumes that a string `<[^<>]+\$[^<>]>` is a reference to an environment variable, whereas anything else involving < or > is redirection, and generally manages to be 99% right. Of course, the problem remains that scripts cannot rely on any Unix tools being available, or that any tools found have Unix-like command line arguments.

Extensions and XS are, in theory, buildable by anyone using free tools. In practice, many don't, as users of the Acorn platform are used to binary distributions. MakeMaker does run, but no available make currently copes with MakeMaker's makefiles; even if and when this should be fixed, the lack of a Unix-like shell will cause problems with makefile rules, especially lines of the form `cd sdbm && make all`, and anything using quoting.

"RISC OS" is the proper name for the operating system, but the value in `$^O` is "riscos" (because we don't like shouting).

Other perls

Perl has been ported to many platforms that do not fit into any of the categories listed above. Some, such as AmigaOS, BeOS, HP MPE/iX, QNX, Plan 9, and VOS, have been well-integrated into the standard Perl source code kit. You may need to see the *ports/* directory on CPAN for information, and possibly binaries, for the likes of: aos, Atari ST, lynxos, riscos, Novell Netware, Tandem Guardian, *etc.* (Yes, we know that some of these OSes may fall under the Unix category, but we are not a standards body.)

Some approximate operating system names and their `$^O` values in the "OTHER" category include:

```
OS            $^O        $Config{'archname'}
--------------------------------------------
Amiga DOS     amigaos    m68k-amigos
BeOS          beos
MPE/iX        mpeix      PA-RISC1.1
```

See also:

- Amiga, *README.amiga* (installed as *perlamiga*).
- Be OS, *README.beos*
- HP 300 MPE/iX, *README.mpeix* and Mark Bixby's web page *http://www.bixby.org/mark/porting.html*
- A free perl5-based PERL.NLM for Novell Netware is available in precompiled binary and source code form from *http://www.novell.com/* as well as from CPAN.
- Plan 9, *README.plan9*

FUNCTION IMPLEMENTATIONS

Listed below are functions that are either completely unimplemented or else have been implemented differently on various platforms. Following each description will be, in parentheses, a list of platforms that the description applies to.

The list may well be incomplete, or even wrong in some places. When in doubt, consult the platform-specific README files in the Perl source distribution, and any other documentation resources accompanying a given port.

Be aware, moreover, that even among Unix-ish systems there are variations.

For many functions, you can also query `%Config`, exported by default from the Config module. For example, to check whether the platform has the `lstat` call, check `$Config{d_lstat}`. See *Config* for a full description of available variables.

Alphabetical Listing of Perl Functions

-X

`-w` only inspects the read-only file attribute (FILE_ATTRIBUTE_READONLY), which determines whether the directory can be deleted, not whether it can be written to. Directories always have read and write access unless denied by discretionary access control lists (DACLs). (Win32)

`-r`, `-w`, `-x`, and `-o` tell whether the file is accessible, which may not reflect UIC-based file protections. (VMS)

`-s` by name on an open file will return the space reserved on disk, rather than the current extent. `-s` on an open filehandle returns the current size. (RISC OS)

`-R`, `-W`, `-X`, `-O` are indistinguishable from `-r`, `-w`, `-x`, `-o`. (Win32, VMS, RISC OS)

`-g`, `-k`, `-l`, `-u`, `-A` are not particularly meaningful. (Win32, VMS, RISC OS)

`-p` is not particularly meaningful. (VMS, RISC OS)

`-d` is true if passed a device spec without an explicit directory. (VMS)

`-x` (or `-X`) determine if a file ends in one of the executable suffixes. `-S` is meaningless. (Win32)

`-x` (or `-X`) determine if a file has an executable file type. (RISC OS)

alarm

Emulated using timers that must be explicitly polled whenever Perl wants to dispatch "safe signals" and therefore cannot interrupt blocking system calls. (Win32)

atan2

Due to issues with various CPUs, math libraries, compilers, and standards, results for `atan2()` may vary depending on any combination of the above. Perl attempts to conform to the Open Group/IEEE standards for the results returned from `atan2()`, but cannot force the issue if the system Perl is run on does not allow it. (Tru64, HP-UX 10.20)

The current version of the standards for `atan2()` is available at *http://www.opengroup.org/onlinepubs/009695399/functions/atan2.html*.

binmode

Meaningless. (RISC OS)

Reopens file and restores pointer; if function fails, underlying filehandle may be closed, or pointer may be in a different position. (VMS)

The value returned by `tell` may be affected after the call, and the filehandle may be flushed. (Win32)

chmod

Only good for changing "owner" read-write access, "group", and "other" bits are meaningless. (Win32)

Only good for changing "owner" and "other" read-write access. (RISC OS)

Access permissions are mapped onto VOS access-control list changes. (VOS)

The actual permissions set depend on the value of the `CYGWIN` in the SYSTEM environment settings. (Cygwin)

chown

Not implemented. (Win32, Plan 9, RISC OS)

Does nothing, but won't fail. (Win32)

A little funky, because VOS's notion of ownership is a little funky (VOS).

chroot

Not implemented. (Win32, VMS, Plan 9, RISC OS, VOS, VM/ESA)

crypt

May not be available if library or source was not provided when building perl. (Win32)

dbmclose

Not implemented. (VMS, Plan 9, VOS)

dbmopen

Not implemented. (VMS, Plan 9, VOS)

dump

Not useful. (RISC OS)

Not supported. (Cygwin, Win32)

Invokes VMS debugger. (VMS)

exec

Implemented via Spawn. (VM/ESA)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

Not supported. (Symbian OS)

exit

Emulates Unix exit() (which considers `exit 1` to indicate an error) by mapping the `1` to SS$_ABORT (`44`). This behavior may be overridden with the pragma `use vmsish 'exit'`. As with the CRTL's exit() function, `exit 0` is also mapped to an exit status of SS$_NORMAL (`1`); this mapping cannot be overridden. Any other argument to exit() is used directly as Perl's exit status. On VMS, unless the future POSIX_EXIT mode is enabled, the exit code should always be a valid VMS exit code and not a generic number. When the POSIX_EXIT mode is enabled, a generic number will be encoded in a method compatible with the C library _POSIX_EXIT macro so that it can be decoded by other programs, particularly ones written in C, like the GNV package. (VMS)

`exit()` resets file pointers, which is a problem when called from a child process (created by `fork()`) in `BEGIN`. A workaround is to use `POSIX::_exit`. (Solaris)

```
    exit unless $Config{archname} =~ /\bsolaris\b/;
    require POSIX and POSIX::_exit(0);
```

fcntl

Not implemented. (Win32)

Some functions available based on the version of VMS. (VMS)

flock

Not implemented (VMS, RISC OS, VOS).

fork

Not implemented. (AmigaOS, RISC OS, VM/ESA, VMS)

Emulated using multiple interpreters. See *perlfork*. (Win32)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

getlogin

Not implemented. (RISC OS)

getpgrp

Not implemented. (Win32, VMS, RISC OS)

getppid

Not implemented. (Win32, RISC OS)

getpriority

Not implemented. (Win32, VMS, RISC OS, VOS, VM/ESA)

getpwnam

Not implemented. (Win32)

Not useful. (RISC OS)

getgrnam

Not implemented. (Win32, VMS, RISC OS)

getnetbyname

Not implemented. (Win32, Plan 9)

getpwuid

Not implemented. (Win32)

Not useful. (RISC OS)

getgrgid

Not implemented. (Win32, VMS, RISC OS)

getnetbyaddr

Not implemented. (Win32, Plan 9)

getprotobynumber

getservbyport

getpwent

Not implemented. (Win32, VM/ESA)

getgrent

Not implemented. (Win32, VMS, VM/ESA)

gethostbyname

`gethostbyname('localhost')` does not work everywhere: you may have to use `gethostbyname('127.0.0.1')`. (Irix 5)

gethostent

Not implemented. (Win32)

getnetent

Not implemented. (Win32, Plan 9)

getprotoent

Not implemented. (Win32, Plan 9)

getservent

Not implemented. (Win32, Plan 9)

sethostent

Not implemented. (Win32, Plan 9, RISC OS)

setnetent

Not implemented. (Win32, Plan 9, RISC OS)

setprotoent

Not implemented. (Win32, Plan 9, RISC OS)

setservent

Not implemented. (Plan 9, Win32, RISC OS)

endpwent

Not implemented. (MPE/iX, VM/ESA, Win32)

endgrent

Not implemented. (MPE/iX, RISC OS, VM/ESA, VMS, Win32)

endhostent

Not implemented. (Win32)

endnetent

Not implemented. (Win32, Plan 9)

endprotoent

Not implemented. (Win32, Plan 9)

endservent

Not implemented. (Plan 9, Win32)

getsockopt SOCKET,LEVEL,OPTNAME

Not implemented. (Plan 9)

glob

This operator is implemented via the File::Glob extension on most platforms. See *File::Glob* for portability information.

gmtime

In theory, gmtime() is reliable from -2**63 to 2**63-1. However, because work arounds in the implementation use floating point numbers, it will become inaccurate as the time gets larger. This is a bug and will be fixed in the future.

On VOS, time values are 32-bit quantities.

ioctl FILEHANDLE,FUNCTION,SCALAR

Not implemented. (VMS)

Available only for socket handles, and it does what the ioctlsocket() call in the Winsock API does. (Win32)

Available only for socket handles. (RISC OS)

kill

Not implemented, hence not useful for taint checking. (RISC OS)

`kill()` doesn't have the semantics of `raise()`, i.e. it doesn't send a signal to the identified process like it does on Unix platforms. Instead `kill($sig, $pid)`

terminates the process identified by $pid, and makes it exit immediately with exit status $sig. As in Unix, if $sig is 0 and the specified process exists, it returns true without actually terminating it. (Win32)

`kill(-9, $pid)` will terminate the process specified by $pid and recursively all child processes owned by it. This is different from the Unix semantics, where the signal will be delivered to all processes in the same process group as the process specified by $pid. (Win32)

Is not supported for process identification number of 0 or negative numbers. (VMS)

link

Not implemented. (MPE/iX, RISC OS, VOS)

Link count not updated because hard links are not quite that hard (They are sort of half-way between hard and soft links). (AmigaOS)

Hard links are implemented on Win32 under NTFS only. They are natively supported on Windows 2000 and later. On Windows NT they are implemented using the Windows POSIX subsystem support and the Perl process will need Administrator or Backup Operator privileges to create hard links.

Available on 64 bit OpenVMS 8.2 and later. (VMS)

localtime

localtime() has the same range as *gmtime*, but because time zone rules change its accuracy for historical and future times may degrade but usually by no more than an hour.

lstat

Not implemented. (RISC OS)

Return values (especially for device and inode) may be bogus. (Win32)

msgctl

msgget

msgsnd

msgrcv

Not implemented. (Win32, VMS, Plan 9, RISC OS, VOS)

open

open to `|-` and `-|` are unsupported. (Win32, RISC OS)

Opening a process does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

readlink

Not implemented. (Win32, VMS, RISC OS)

rename

Can't move directories between directories on different logical volumes. (Win32)

rewinddir

Will not cause readdir() to re-read the directory stream. The entries already read before the rewinddir() call will just be returned again from a cache buffer. (Win32)

select

Only implemented on sockets. (Win32, VMS)

Only reliable on sockets. (RISC OS)

Note that the `select FILEHANDLE` form is generally portable.

semctl

semget

semop

Not implemented. (Win32, VMS, RISC OS)

setgrent

Not implemented. (MPE/iX, VMS, Win32, RISC OS)

setpgrp

Not implemented. (Win32, VMS, RISC OS, VOS)

setpriority

Not implemented. (Win32, VMS, RISC OS, VOS)

setpwent

Not implemented. (MPE/iX, Win32, RISC OS)

setsockopt

Not implemented. (Plan 9)

shmctl

shmget

shmread

shmwrite

Not implemented. (Win32, VMS, RISC OS, VOS)

sockatmark

A relatively recent addition to socket functions, may not be implemented even in Unix platforms.

socketpair

Not implemented. (RISC OS, VM/ESA)

Available on OpenVOS Release 17.0 or later. (VOS)

Available on 64 bit OpenVMS 8.2 and later. (VMS)

stat

Platforms that do not have rdev, blksize, or blocks will return these as '', so numeric comparison or manipulation of these fields may cause 'not numeric' warnings.

ctime not supported on UFS (Mac OS X).

ctime is creation time instead of inode change time (Win32).

device and inode are not meaningful. (Win32)

device and inode are not necessarily reliable. (VMS)

mtime, atime and ctime all return the last modification time. Device and inode are not necessarily reliable. (RISC OS)

dev, rdev, blksize, and blocks are not available. inode is not meaningful and will differ between stat calls on the same file. (os2)

some versions of cygwin when doing a stat("foo") and if not finding it may then attempt to stat("foo.exe") (Cygwin)

On Win32 stat() needs to open the file to determine the link count and update

attributes that may have been changed through hard links. Setting ${^WIN32_SLOPPY_STAT} to a true value speeds up stat() by not performing this operation. (Win32)

symlink

Not implemented. (Win32, RISC OS)

Implemented on 64 bit VMS 8.3. VMS requires the symbolic link to be in Unix syntax if it is intended to resolve to a valid path.

syscall

Not implemented. (Win32, VMS, RISC OS, VOS, VM/ESA)

sysopen

The traditional "0", "1", and "2" MODEs are implemented with different numeric values on some systems. The flags exported by `Fcntl` (O_RDONLY, O_WRONLY, O_RDWR) should work everywhere though. (Mac OS, OS/390, VM/ESA)

system

As an optimization, may not call the command shell specified in `$ENV{PERL5SHELL}`. `system(1, @args)` spawns an external process and immediately returns its process designator, without waiting for it to terminate. Return value may be used subsequently in `wait` or `waitpid`. Failure to spawn() a subprocess is indicated by setting $? to "255 << 8". $? is set in a way compatible with Unix (i.e. the exitstatus of the subprocess is obtained by "$? >> 8", as described in the documentation). (Win32)

There is no shell to process metacharacters, and the native standard is to pass a command line terminated by "\n" "\r" or "\0" to the spawned program. Redirection such as `> foo` is performed (if at all) by the run time library of the spawned program. `system` *list* will call the Unix emulation library's `exec` emulation, which attempts to provide emulation of the stdin, stdout, stderr in force in the parent, providing the child program uses a compatible version of the emulation library. *scalar* will call the native command line direct and no such emulation of a child Unix program will exists. Mileage **will** vary. (RISC OS)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

The return value is POSIX-like (shifted up by 8 bits), which only allows room for a made-up value derived from the severity bits of the native 32-bit condition code (unless overridden by `use vmsish 'status'`). If the native condition code is one that has a POSIX value encoded, the POSIX value will be decoded to extract the expected exit value. For more details see *"$?" in perlvms*. (VMS)

times

"cumulative" times will be bogus. On anything other than Windows NT or Windows 2000, "system" time will be bogus, and "user" time is actually the time returned by the clock() function in the C runtime library. (Win32)

Not useful. (RISC OS)

truncate

Not implemented. (Older versions of VMS)

Truncation to same-or-shorter lengths only. (VOS)

If a FILEHANDLE is supplied, it must be writable and opened in append mode (i.e., use `open(FH, '>>filename')` or `sysopen(FH,...,O_APPEND|O_RDWR)`. If a filename is supplied, it should not be held open elsewhere. (Win32)

umask

Returns undef where unavailable, as of version 5.005.

`umask` works but the correct permissions are set only when the file is finally closed. (AmigaOS)

utime

Only the modification time is updated. (BeOS, VMS, RISC OS)

May not behave as expected. Behavior depends on the C runtime library's implementation of utime(), and the filesystem being used. The FAT filesystem typically does not support an "access time" field, and it may limit timestamps to a granularity of two seconds. (Win32)

wait

waitpid

Can only be applied to process handles returned for processes spawned using `system(1, ...)` or pseudo processes created with `fork()`. (Win32)

Not useful. (RISC OS)

Supported Platforms

The following platforms are known to build Perl 5.12 (as of April 2010, its release date) from the standard source code distribution available at *http://www.cpan.org/src*

Linux (x86, ARM, IA64)

HP-UX

AIX

Win32

- Windows 2000
- Windows XP
- Windows Server 2003
- Windows Vista
- Windows Server 2008
- Windows 7

Cygwin

Solaris (x86, SPARC)

OpenVMS

- Alpha (7.2 and later)
- I64 (8.2 and later)

Symbian

NetBSD

FreeBSD

Debian GNU/kFreeBSD

Haiku

Irix (6.5. What else?)

OpenBSD

Dragonfly BSD

QNX Neutrino RTOS (6.5.0)

MirOS BSD

Caveats:

time_t issues that may or may not be fixed

Symbian (Series 60 v3, 3.2 and 5 - what else?)

Stratus VOS / OpenVOS

AIX

EOL Platforms (Perl 5.14)

The following platforms were supported by a previous version of Perl but have been officially removed from Perl's source code as of 5.12:

Atari MiNT

Apollo Domain/OS

Apple Mac OS 8/9

Tenon Machten

The following platforms were supported up to 5.10. They may still have worked in 5.12, but supporting code has been removed for 5.14:

Windows 95

Windows 98

Windows ME

Windows NT4

Supported Platforms (Perl 5.8)

As of July 2002 (the Perl release 5.8.0), the following platforms were able to build Perl from the standard source code distribution available at *http://www.cpan.org/src/*

```
AIX
BeOS
BSD/OS          (BSDi)
Cygwin
DG/UX
DOS DJGPP       1)
DYNIX/ptx
EPOC R5
FreeBSD
HI-UXMPP        (Hitachi) (5.8.0 worked but we didn't know it)
HP-UX
IRIX
Linux
Mac OS Classic
Mac OS X        (Darwin)
MPE/iX
NetBSD
NetWare
NonStop-UX
ReliantUNIX     (formerly SINIX)
OpenBSD
OpenVMS         (formerly VMS)
Open UNIX       (Unixware) (since Perl 5.8.1/5.9.0)
OS/2
OS/400          (using the PASE) (since Perl 5.8.1/5.9.0)
PowerUX
```

```
POSIX-BC        (formerly BS2000)
QNX
Solaris
SunOS 4
SUPER-UX        (NEC)
Tru64 UNIX      (formerly DEC OSF/1, Digital UNIX)
UNICOS
UNICOS/mk
UTS
VOS
Win95/98/ME/2K/XP 2)
WinCE
z/OS            (formerly OS/390)
VM/ESA

1) in DOS mode either the DOS or OS/2 ports can be used
2) compilers: Borland, MinGW (GCC), VC6
```

The following platforms worked with the previous releases (5.6 and 5.7), but we did not manage either to fix or to test these in time for the 5.8.0 release. There is a very good chance that many of these will work fine with the 5.8.0.

```
BSD/OS
DomainOS
Hurd
LynxOS
MachTen
PowerMAX
SCO SV
SVR4
Unixware
Windows 3.1
```

Known to be broken for 5.8.0 (but 5.6.1 and 5.7.2 can be used):

```
AmigaOS
```

The following platforms have been known to build Perl from source in the past (5.005_03 and earlier), but we haven't been able to verify their status for the current release, either because the hardware/software platforms are rare or because we don't have an active champion on these platforms--or both. They used to work, though, so go ahead and try compiling them, and let perlbug@perl.org of any trouble.

```
3b1
A/UX
ConvexOS
CX/UX
DC/OSx
DDE SMES
DOS EMX
Dynix
EP/IX
ESIX
FPS
GENIX
Greenhills
```

```
ISC
MachTen 68k
MPC
NEWS-OS
NextSTEP
OpenSTEP
Opus
Plan 9
RISC/os
SCO ODT/OSR
Stellar
SVR2
TI1500
TitanOS
Ultrix
Unisys Dynix
```

The following platforms have their own source code distributions and binaries available via *http://www.cpan.org/ports/*

	Perl release
OS/400 (ILE)	5.005_02
Tandem Guardian	5.004

The following platforms have only binaries available via *http://www.cpan.org/ports/index.html* :

	Perl release
Acorn RISCOS	5.005_02
AOS	5.002
LynxOS	5.004_02

Although we do suggest that you always build your own Perl from the source code, both for maximal configurability and for security, in case you are in a hurry you can check *http://www.cpan.org/ports/index.html* for binary distributions.

SEE ALSO

perlaix, *perlamiga*, *perlapollo*, *perlbeos*, *perlbs2000*, *perlce*, *perlcygwin*, *perldgux*, *perldos*, *perlepoc*, *perlebcdic*, *perlfreebsd*, *perlhurd*, *perlhpux*, *perlirix*, *perlmacos*, *perlmacosx*, *perlmpeix*, *perlnetware*, *perlos2*, *perlos390*, *perlos400*, *perlplan9*, *perlqnx*, *perlsolaris*, *perltru64*, *perlunicode*, *perlvmesa*, *perlvms*, *perlvos*, *perlwin32*, and *Win32*.

AUTHORS / CONTRIBUTORS

Abigail <abigail@foad.org>, Charles Bailey <bailey@newman.upenn.edu>, Graham Barr <gbarr@pobox.com>, Tom Christiansen <tchrist@perl.com>, Nicholas Clark <nick@ccl4.org>, Thomas Dorner <Thomas.Dorner@start.de>, Andy Dougherty <doughera@lafayette.edu>, Dominic Dunlop <domo@computer.org>, Neale Ferguson <neale@vma.tabnsw.com.au>, David J. Fiander <davidf@mks.com>, Paul Green <Paul.Green@stratus.com>, M.J.T. Guy <mjtg@cam.ac.uk>, Jarkko Hietaniemi <jhi@iki.fi>, Luther Huffman <lutherh@stratcom.com>, Nick Ing-Simmons <nick@ing-simmons.net>, Andreas J. König <a.koenig@mind.de>, Markus Laker <mlaker@contax.co.uk>, Andrew M. Langmead <aml@world.std.com>, Larry Moore <ljmoore@freespace.net>, Paul Moore <Paul.Moore@uk.origin-it.com>, Chris Nandor <pudge@pobox.com>, Matthias Neeracher <neeracher@mac.com>, Philip Newton <pne@cpan.org>, Gary Ng <71564.1743@CompuServe.COM>, Tom Phoenix

<rootbeer@teleport.com>, André Pirard <A.Pirard@ulg.ac.be>, Peter Prymmer <pvhp@forte.com>, Hugo van der Sanden <hv@crypt0.demon.co.uk>, Gurusamy Sarathy <gsar@activestate.com>, Paul J. Schinder <schinder@pobox.com>, Michael G Schwern <schwern@pobox.com>, Dan Sugalski <dan@sidhe.org>, Nathan Torkington <gnat@frii.com>, John Malmberg <wb8tyw@qsl.net>