NAME

perlvms - VMS-specific documentation for Perl

DESCRIPTION

Gathered below are notes describing details of Perl 5's behavior on VMS. They are a supplement to the regular Perl 5 documentation, so we have focussed on the ways in which Perl 5 functions differently under VMS than it does under Unix, and on the interactions between Perl and the rest of the operating system. We haven't tried to duplicate complete descriptions of Perl features from the main Perl documentation, which can be found in the *[.pod]* subdirectory of the Perl distribution.

We hope these notes will save you from confusion and lost sleep when writing Perl scripts on VMS. If you find we've missed something you think should appear here, please don't hesitate to drop a line to vmsperl@perl.org.

Installation

Directions for building and installing Perl 5 can be found in the file *README.vms* in the main source directory of the Perl distribution..

Organization of Perl Images

Core Images

During the installation process, three Perl images are produced. *Miniperl.Exe* is an executable image which contains all of the basic functionality of Perl, but cannot take advantage of Perl extensions. It is used to generate several files needed to build the complete Perl and various extensions. Once you've finished installing Perl, you can delete this image.

Most of the complete Perl resides in the shareable image *PerlShr.Exe*, which provides a core to which the Perl executable image and all Perl extensions are linked. You should place this image in *Sys$Share*, or define the logical name *PerlShr* to translate to the full file specification of this image. It should be world readable. (Remember that if a user has execute only access to *PerlShr*, VMS will treat it as if it were a privileged shareable image, and will therefore require all downstream shareable images to be INSTALLed, etc.)

Finally, *Perl.Exe* is an executable image containing the main entry point for Perl, as well as some initialization code. It should be placed in a public directory, and made world executable. In order to run Perl with command line arguments, you should define a foreign command to invoke this image.

Perl Extensions

Perl extensions are packages which provide both XS and Perl code to add new functionality to perl. (XS is a meta-language which simplifies writing C code which interacts with Perl, see *perlxs* for more details.) The Perl code for an extension is treated like any other library module - it's made available in your script through the appropriate `use` or `require` statement, and usually defines a Perl package containing the extension.

The portion of the extension provided by the XS code may be connected to the rest of Perl in either of two ways. In the **static** configuration, the object code for the extension is linked directly into *PerlShr.Exe*, and is initialized whenever Perl is invoked. In the **dynamic** configuration, the extension's machine code is placed into a separate shareable image, which is mapped by Perl's DynaLoader when the extension is `used` or `required` in your script. This allows you to maintain the extension as a separate entity, at the cost of keeping track of the additional shareable image. Most extensions can be set up as either static or dynamic.

The source code for an extension usually resides in its own directory. At least three files are generally provided: *Extshortname.xs* (where *Extshortname* is the portion of the extension's name following the last `::`), containing the XS code, *Extshortname.pm*, the Perl library module for the extension, and *Makefile.PL*, a Perl script which uses the `MakeMaker` library modules supplied with Perl to generate a *Descrip.MMS* file for the extension.

Installing static extensions

Since static extensions are incorporated directly into *PerlShr.Exe*, you'll have to rebuild Perl to incorporate a new extension. You should edit the main *Descrip.MMS* or *Makefile* you use to build Perl, adding the extension's name to the `ext` macro, and the extension's object file to the `extobj` macro. You'll also need to build the extension's object file, either by adding dependencies to the main *Descrip.MMS*, or using a separate *Descrip.MMS* for the extension. Then, rebuild *PerlShr.Exe* to incorporate the new code.

Finally, you'll need to copy the extension's Perl library module to the *[.Extname]* subdirectory under one of the directories in `@INC`, where *Extname* is the name of the extension, with all `::` replaced by `.` (e.g. the library module for extension Foo::Bar would be copied to a *[.Foo.Bar]* subdirectory).

Installing dynamic extensions

In general, the distributed kit for a Perl extension includes a file named Makefile.PL, which is a Perl program which is used to create a *Descrip.MMS* file which can be used to build and install the files required by the extension. The kit should be unpacked into a directory tree **not** under the main Perl source directory, and the procedure for building the extension is simply

```
$ perl Makefile.PL  ! Create Descrip.MMS
$ mmk               ! Build necessary files
$ mmk test          ! Run test code, if supplied
$ mmk install       ! Install into public Perl tree
```

N.B. The procedure by which extensions are built and tested creates several levels (at least 4) under the directory in which the extension's source files live. For this reason if you are running a version of VMS prior to V7.1 you shouldn't nest the source directory too deeply in your directory structure lest you exceed RMS' maximum of 8 levels of subdirectory in a filespec. (You can use rooted logical names to get another 8 levels of nesting, if you can't place the files near the top of the physical directory structure.)

VMS support for this process in the current release of Perl is sufficient to handle most extensions. However, it does not yet recognize extra libraries required to build shareable images which are part of an extension, so these must be added to the linker options file for the extension by hand. For instance, if the *PGPLOT* extension to Perl requires the *PGPLOTSHR.EXE* shareable image in order to properly link the Perl extension, then the line `PGPLOTSHR/Share` must be added to the linker options file *PGPLOT.Opt* produced during the build process for the Perl extension.

By default, the shareable image for an extension is placed in the *[.lib.site_perl.autoArch.Extname]* directory of the installed Perl directory tree (where *Arch* is *VMS_VAX* or *VMS_AXP*, and *Extname* is the name of the extension, with each `::` translated to `.`). (See the MakeMaker documentation for more details on installation options for extensions.) However, it can be manually placed in any of several locations:

- the *[.Lib.Auto.Arch$PVersExtname]* subdirectory of one of the directories in `@INC` (where *PVers* is the version of Perl you're using, as supplied in `$]`, with '.' converted to '_'), or
- one of the directories in `@INC`, or
- a directory which the extensions Perl library module passes to the DynaLoader when asking it to map the shareable image, or
- *Sys$Share* or *Sys$Library*.

If the shareable image isn't in any of these places, you'll need to define a logical name *Extshortname*, where *Extshortname* is the portion of the extension's name after the last `::`, which translates to the full file specification of the shareable image.

File specifications

Syntax

We have tried to make Perl aware of both VMS-style and Unix- style file specifications wherever possible. You may use either style, or both, on the command line and in scripts, but you may not combine the two styles within a single file specification. VMS Perl interprets Unix pathnames in much the same way as the CRTL (*e.g.* the first component of an absolute path is read as the device name for the VMS file specification). There are a set of functions provided in the `VMS::Filespec` package for explicit interconversion between VMS and Unix syntax; its documentation provides more details.

Perl is now in the process of evolving to follow the setting of the DECC$* feature logical names in the interpretation of UNIX pathnames. This is still a work in progress.

For handling extended characters, and case sensitivity, as long as DECC$POSIX_COMPLIANT_PATHNAMES, DECC$FILENAME_UNIX_REPORT, and DECC$FILENAME_UNIX_ONLY are not set, then the older Perl behavior for conversions of file specifications from UNIX to VMS is followed, except that VMS paths with concealed rooted logical names are now translated correctly to UNIX paths.

With those features set, then new routines may handle the translation, because some of the rules are different. The presence of ./.../ in a UNIX path is no longer translated to the VMS [...]. It will translate to [.^.^.^.]. To be compatible with what MakeMaker expects, if a VMS path can not be translated to a UNIX path when unixify is called, it is passed through unchanged. So unixify("[...]") will return "[...]".

The handling of extended characters will also be better with the newer translation routines. But more work is needed to fully support extended file syntax names. In particular, at this writing Pathtools can not deal with directories containing some extended characters.

There are several ambiguous cases where a conversion routine can not determine if an input filename is in UNIX format or in VMS format, since now both VMS UNIX file specifications can have characters in them that could be mistaken for syntax delimiters of the other type. So some pathnames simply can not be used in a mode that allows either type of pathname to be present.

Perl will tend to assume that an ambiguous filename is in UNIX format.

Allowing "." as a version delimiter is simply incompatible with determining if a pathname is already VMS format or UNIX with the extended file syntax. There is no way to know if "perl-5.8.6" that TAR produces is a UNIX "perl-5.8.6" or a VMS "perl-5.8;6" when passing it to unixify() or vmsify().

The DECC$FILENAME_UNIX_REPORT or the DECC$FILENAME_UNIX_ONLY logical names control how Perl interprets filenames.

The DECC$FILENAME_UNIX_ONLY setting has not been tested at this time. Perl uses traditional OpenVMS file specifications internally and in the test harness, so this mode may have limited use, or require more changes to make usable.

Everything about DECC$FILENAME_UNIX_REPORT should be assumed to apply to DECC$FILENAME_UNIX_ONLY mode. The DECC$FILENAME_UNIX_ONLY differs in that it expects all filenames passed to the C runtime to be already in UNIX format.

Again, currently most of the core Perl modules have not yet been updated to understand that VMS is not as limited as it use to be. Fixing that is a work in progress.

The logical name DECC$POSIX_COMPLIANT_PATHNAMES is new with the RMS Symbolic Link SDK. This version of Perl does not support it being set.

Filenames are case-insensitive on VAX, and on ODS-2 formatted volumes on ALPHA and I64.

On ODS-5 volumes filenames are case preserved and on newer versions of OpenVMS can be optionally case sensitive.

On ALPHA and I64, Perl is in the process of being changed to follow the process case sensitivity setting to report if the file system is case sensitive.

Perl programs should not assume that VMS is case blind, or that filenames will be in lowercase.

Programs should use the File::Spec:case_tolerant setting to determine the state, and not the $^O setting.

For consistency, when the above feature is clear and when not otherwise overridden by DECC feature logical names, most Perl routines return file specifications using lower case letters only, regardless of the case used in the arguments passed to them. (This is true only when running under VMS; Perl respects the case-sensitivity of OSs like Unix.)

We've tried to minimize the dependence of Perl library modules on Unix syntax, but you may find that some of these, as well as some scripts written for Unix systems, will require that you use Unix syntax, since they will assume that '/' is the directory separator, *etc.* If you find instances of this in the Perl distribution itself, please let us know, so we can try to work around them.

Also when working on Perl programs on VMS, if you need a syntax in a specific operating system format, then you need to either check the appropriate DECC$ feature logical, or call a conversion routine to force it to that format.

Wildcard expansion

File specifications containing wildcards are allowed both on the command line and within Perl globs (e.g. `<*.c>`). If the wildcard filespec uses VMS syntax, the resultant filespecs will follow VMS syntax; if a Unix-style filespec is passed in, Unix-style filespecs will be returned. Similar to the behavior of wildcard globbing for a Unix shell, one can escape command line wildcards with double quotation marks " around a perl program command line argument. However, owing to the stripping of " characters carried out by the C handling of argv you will need to escape a construct such as this one (in a directory containing the files *PERL.C*, *PERL.EXE*, *PERL.H*, and *PERL.OBJ*):

```
$ perl -e "print join(' ',@ARGV)" perl.*
perl.c perl.exe perl.h perl.obj
```

in the following triple quoted manner:

```
$ perl -e "print join(' ',@ARGV)" """perl.*"""
perl.*
```

In both the case of unquoted command line arguments or in calls to `glob()` VMS wildcard expansion is performed. (csh-style wildcard expansion is available if you use `File::Glob::glob`.) If the wildcard filespec contains a device or directory specification, then the resultant filespecs will also contain a device and directory; otherwise, device and directory information are removed. VMS-style resultant filespecs will contain a full device and directory, while Unix-style resultant filespecs will contain only as much of a directory path as was present in the input filespec. For example, if your default directory is Perl_Root:[000000], the expansion of `[.t]*.*` will yield filespecs like "perl_root:[t]base.dir", while the expansion of `t/*/*` will yield filespecs like "t/base.dir". (This is done to match the behavior of glob expansion performed by Unix shells.)

Similarly, the resultant filespec will contain the file version only if one was present in the input filespec.

Pipes

Input and output pipes to Perl filehandles are supported; the "file name" is passed to lib$spawn() for asynchronous execution. You should be careful to close any pipes you have opened in a Perl script, lest you leave any "orphaned" subprocesses around when Perl exits.

You may also use backticks to invoke a DCL subprocess, whose output is used as the return value of the expression. The string between the backticks is handled as if it were the argument to the `system`

operator (see below). In this case, Perl will wait for the subprocess to complete before continuing.

The mailbox (MBX) that perl can create to communicate with a pipe defaults to a buffer size of 512. The default buffer size is adjustable via the logical name PERL_MBX_SIZE provided that the value falls between 128 and the SYSGEN parameter MAXBUF inclusive. For example, to double the MBX size from the default within a Perl program, use `$ENV{'PERL_MBX_SIZE'} = 1024;` and then open and use pipe constructs. An alternative would be to issue the command:

```
$ Define PERL_MBX_SIZE 1024
```

before running your wide record pipe program. A larger value may improve performance at the expense of the BYTLM UAF quota.

PERL5LIB and PERLLIB

The PERL5LIB and PERLLIB logical names work as documented in *perl*, except that the element separator is '|' instead of ':'. The directory specifications may use either VMS or Unix syntax.

The Perl Forked Debugger

The Perl forked debugger places the debugger commands and output in a separate X-11 terminal window so that commands and output from multiple processes are not mixed together.

Perl on VMS supports an emulation of the forked debugger when Perl is run on a VMS system that has X11 support installed.

To use the forked debugger, you need to have the default display set to an X-11 Server and some environment variables set that Unix expects.

The forked debugger requires the environment variable `TERM` to be `xterm`, and the environment variable `DISPLAY` to exist. `xterm` must be in lower case.

```
$define TERM "xterm"

$define DISPLAY "hostname:0.0"
```

Currently the value of `DISPLAY` is ignored. It is recommended that it be set to be the hostname of the display, the server and screen in UNIX notation. In the future the value of DISPLAY may be honored by Perl instead of using the default display.

It may be helpful to always use the forked debugger so that script I/O is separated from debugger I/O. You can force the debugger to be forked by assigning a value to the logical name <PERLDB_PIDS> that is not a process identification number.

```
$define PERLDB_PIDS XXXX
```

PERL_VMS_EXCEPTION_DEBUG

The PERL_VMS_EXCEPTION_DEBUG being defined as "ENABLE" will cause the VMS debugger to be invoked if a fatal exception that is not otherwise handled is raised. The purpose of this is to allow debugging of internal Perl problems that would cause such a condition.

This allows the programmer to look at the execution stack and variables to find out the cause of the exception. As the debugger is being invoked as the Perl interpreter is about to do a fatal exit, continuing the execution in debug mode is usally not practical.

Starting Perl in the VMS debugger may change the program execution profile in a way that such problems are not reproduced.

The `kill` function can be used to test this functionality from within a program.

In typical VMS style, only the first letter of the value of this logical name is actually checked in a case insensitive mode, and it is considered enabled if it is the value "T","1" or "E".

This logical name must be defined before Perl is started.

Command line

I/O redirection and backgrounding

Perl for VMS supports redirection of input and output on the command line, using a subset of Bourne shell syntax:

- `<file` reads stdin from `file`,
- `>file` writes stdout to `file`,
- `>>file` appends stdout to `file`,
- `2>file` writes stderr to `file`,
- `2>>file` appends stderr to `file`, and
- `2>&1` redirects stderr to stdout.

In addition, output may be piped to a subprocess, using the character '|'. Anything after this character on the command line is passed to a subprocess for execution; the subprocess takes the output of Perl as its input.

Finally, if the command line ends with '&', the entire command is run in the background as an asynchronous subprocess.

Command line switches

The following command line switches behave differently under VMS than described in *perlrun*. Note also that in order to pass uppercase switches to Perl, you need to enclose them in double-quotes on the command line, since the CRTL downcases all unquoted strings.

On newer 64 bit versions of OpenVMS, a process setting now controls if the quoting is needed to preserve the case of command line arguments.

-i

If the `-i` switch is present but no extension for a backup copy is given, then inplace editing creates a new version of a file; the existing copy is not deleted. (Note that if an extension is given, an existing file is renamed to the backup file, as is the case under other operating systems, so it does not remain as a previous version under the original filename.)

-S

If the `"-S"` or `-"S"` switch is present *and* the script name does not contain a directory, then Perl translates the logical name DCL$PATH as a searchlist, using each translation as a directory in which to look for the script. In addition, if no file type is specified, Perl looks in each directory for a file matching the name specified, with a blank type, a type of *.pl*, and a type of *.com*, in that order.

-u

The `-u` switch causes the VMS debugger to be invoked after the Perl program is compiled, but before it has run. It does not create a core dump file.

Perl functions

As of the time this document was last revised, the following Perl functions were implemented in the VMS port of Perl (functions marked with * are discussed in more detail below):

```
file tests*, abs, alarm, atan, backticks*, binmode*, bless,
```

```
caller, chdir, chmod, chown, chomp, chop, chr,
close, closedir, cos, crypt*, defined, delete,
die, do, dump*, each, endpwent, eof, eval, exec*,
exists, exit, exp, fileno, getc, getlogin, getppid,
getpwent*, getpwnam*, getpwuid*, glob, gmtime*, goto,
grep, hex, import, index, int, join, keys, kill*,
last, lc, lcfirst, length, local, localtime, log, m//,
map, mkdir, my, next, no, oct, open, opendir, ord, pack,
pipe, pop, pos, print, printf, push, q//, qq//, qw//,
qx//*, quotemeta, rand, read, readdir, redo, ref, rename,
require, reset, return, reverse, rewinddir, rindex,
rmdir, s///, scalar, seek, seekdir, select(internal),
select (system call)*, setpwent, shift, sin, sleep,
sort, splice, split, sprintf, sqrt, srand, stat,
study, substr, sysread, system*, syswrite, tell,
telldir, tie, time, times*, tr///, uc, ucfirst, umask,
undef, unlink*, unpack, untie, unshift, use, utime*,
values, vec, wait, waitpid*, wantarray, warn, write, y///
```

The following functions were not implemented in the VMS port, and calling them produces a fatal error (usually) or undefined behavior (rarely, we hope):

```
chroot, dbmclose, dbmopen, flock, fork*,
getpgrp, getpriority, getgrent, getgrgid,
getgrnam, setgrent, endgrent, ioctl, link, lstat,
msgctl, msgget, msgsend, msgrcv, readlink, semctl,
semget, semop, setpgrp, setpriority, shmctl, shmget,
shmread, shmwrite, socketpair, symlink, syscall
```

The following functions are available on Perls compiled with Dec C 5.2 or greater and running VMS 7.0 or greater:

```
truncate
```

The following functions are available on Perls built on VMS 7.2 or greater:

```
fcntl (without locking)
```

The following functions may or may not be implemented, depending on what type of socket support you've built into your copy of Perl:

```
accept, bind, connect, getpeername,
gethostbyname, getnetbyname, getprotobyname,
getservbyname, gethostbyaddr, getnetbyaddr,
getprotobynumber, getservbyport, gethostent,
getnetent, getprotoent, getservent, sethostent,
setnetent, setprotoent, setservent, endhostent,
endnetent, endprotoent, endservent, getsockname,
getsockopt, listen, recv, select(system call)*,
send, setsockopt, shutdown, socket
```

The following function is available on Perls built on 64 bit OpenVMS 8.2 with hard links enabled on an ODS-5 formatted build disk. If someone with an OpenVMS 7.3-1 system were to modify configure.com and test the results, this feature can be brought back to OpenVMS 7.3-1 and later. Hardlinks must be enabled on the build disk because if the build procedure sees this feature enabled, it uses it.

```
link
```

The following functions are available on Perls built on 64 bit OpenVMS 8.2 and can be implemented on OpenVMS 7.3-2 if someone were to modify configure.com and test the results. (While in the build, at the time of this writing, they have not been specifically tested.)

```
getgrgid, getgrnam, getpwnam, getpwuid,
setgrent, ttyname
```

The following functions are available on Perls built on 64 bit OpenVMS 8.2 and later. (While in the build, at the time of this writing, they have not been specifically tested.)

```
statvfs, socketpair
```

The following functions are expected to soon be available on Perls built on 64 bit OpenVMS 8.2 or later with the RMS Symbolic link package. Use of symbolic links at this time effectively requires the DECC$POSIX_COMPLIANT_PATHNAMES to defined as 3, and operating in a DECC$FILENAME_UNIX_REPORT mode.

```
lchown, link, lstat, readlink, symlink
```

File tests

The tests `-b`, `-B`, `-c`, `-C`, `-d`, `-e`, `-f`, `-o`, `-M`, `-s`, `-S`, `-t`, `-T`, and `-z` work as advertised. The return values for `-r`, `-w`, and `-x` tell you whether you can actually access the file; this may not reflect the UIC-based file protections. Since real and effective UIC don't differ under VMS, `-O`, `-R`, `-W`, and `-X` are equivalent to `-o`, `-r`, `-w`, and `-x`. Similarly, several other tests, including `-A`, `-g`, `-k`, `-l`, `-p`, and `-u`, aren't particularly meaningful under VMS, and the values returned by these tests reflect whatever your CRTL `stat()` routine does to the equivalent bits in the st_mode field. Finally, `-d` returns true if passed a device specification without an explicit directory (e.g. `DUA1:`), as well as if passed a directory.

There are DECC feature logical names AND ODS-5 volume attributes that also control what values are returned for the date fields.

Note: Some sites have reported problems when using the file-access tests (`-r`, `-w`, and `-x`) on files accessed via DEC's DFS. Specifically, since DFS does not currently provide access to the extended file header of files on remote volumes, attempts to examine the ACL fail, and the file tests will return false, with `$!` indicating that the file does not exist. You can use `stat` on these files, since that checks UIC-based protection only, and then manually check the appropriate bits, as defined by your C compiler's *stat.h*, in the mode value it returns, if you need an approximation of the file's protections.

backticks

Backticks create a subprocess, and pass the enclosed string to it for execution as a DCL command. Since the subprocess is created directly via `lib$spawn()`, any valid DCL command string may be specified.

binmode FILEHANDLE

The `binmode` operator will attempt to insure that no translation of carriage control occurs on input from or output to this filehandle. Since this involves reopening the file and then restoring its file position indicator, if this function returns FALSE, the underlying filehandle may no longer point to an open file, or may point to a different position in the file than before `binmode` was called.

Note that `binmode` is generally not necessary when using normal filehandles; it is provided so that you can control I/O to existing record-structured files when necessary. You can also use the `vmsfopen` function in the VMS::Stdio extension to gain finer control of I/O to files and

devices with different record structures.

crypt PLAINTEXT, USER

The `crypt` operator uses the `sys$hash_password` system service to generate the hashed representation of PLAINTEXT. If USER is a valid username, the algorithm and salt values are taken from that user's UAF record. If it is not, then the preferred algorithm and a salt of 0 are used. The quadword encrypted value is returned as an 8-character string.

The value returned by `crypt` may be compared against the encrypted password from the UAF returned by the `getpw*` functions, in order to authenticate users. If you're going to do this, remember that the encrypted password in the UAF was generated using uppercase username and password strings; you'll have to upcase the arguments to `crypt` to insure that you'll get the proper value:

```
sub validate_passwd {
    my($user,$passwd) = @_;
    my($pwdhash);
    if ( !($pwdhash = (getpwnam($user))[1]) ||
           $pwdhash ne crypt("\U$passwd","\U$name") ) {
        intruder_alert($name);
    }
    return 1;
}
```

die

`die` will force the native VMS exit status to be an SS$_ABORT code if neither of the $! or $? status values are ones that would cause the native status to be interpreted as being what VMS classifies as SEVERE_ERROR severity for DCL error handling.

When the future POSIX_EXIT mode is active, `die`, the native VMS exit status value will have either one of the `$!` or $? or `$^E` or the UNIX value 255 encoded into it in a way that the effective original value can be decoded by other programs written in C, including Perl and the GNV package. As per the normal non-VMS behavior of `die` if either `$!` or $? are non-zero, one of those values will be encoded into a native VMS status value. If both of the UNIX status values are 0, and the `$^E` value is set one of ERROR or SEVERE_ERROR severity, then the `$^E` value will be used as the exit code as is. If none of the above apply, the UNIX value of 255 will be encoded into a native VMS exit status value.

Please note a significant difference in the behavior of `die` in the future POSIX_EXIT mode is that it does not force a VMS SEVERE_ERROR status on exit. The UNIX exit values of 2 through 255 will be encoded in VMS status values with severity levels of SUCCESS. The UNIX exit value of 1 will be encoded in a VMS status value with a severity level of ERROR. This is to be compatible with how the VMS C library encodes these values.

The minimum severity level set by `die` in a future POSIX_EXIT mode may be changed to be ERROR or higher before that mode becomes fully active depending on the results of testing and further review. If this is done, the behavior of c<DIE> in the future POSIX_EXIT will close enough to the default mode that most DCL shell scripts will probably not notice a difference.

See $? for a description of the encoding of the UNIX value to produce a native VMS status containing it.

dump

Rather than causing Perl to abort and dump core, the `dump` operator invokes the VMS debugger. If you continue to execute the Perl program under the debugger, control will be transferred to the label specified as the argument to `dump`, or, if no label was specified, back to the beginning of the program. All other state of the program (*e.g.* values of variables, open file handles) are not affected by calling `dump`.

exec LIST

A call to `exec` will cause Perl to exit, and to invoke the command given as an argument to `exec` via `lib$do_command`. If the argument begins with '@' or '$' (other than as part of a filespec), then it is executed as a DCL command. Otherwise, the first token on the command line is treated as the filespec of an image to run, and an attempt is made to invoke it (using *.Exe* and the process defaults to expand the filespec) and pass the rest of `exec`'s argument to it as parameters. If the token has no file type, and matches a file with null type, then an attempt is made to determine whether the file is an executable image which should be invoked using `MCR` or a text file which should be passed to DCL as a command procedure.

fork

While in principle the `fork` operator could be implemented via (and with the same rather severe limitations as) the CRTL `vfork()` routine, and while some internal support to do just that is in place, the implementation has never been completed, making `fork` currently unavailable. A true kernel `fork()` is expected in a future version of VMS, and the pseudo-fork based on interpreter threads may be available in a future version of Perl on VMS (see *perlfork*). In the meantime, use `system`, backticks, or piped filehandles to create subprocesses.

getpwent

getpwnam

getpwuid

These operators obtain the information described in *perlfunc*, if you have the privileges necessary to retrieve the named user's UAF information via `sys$getuai`. If not, then only the `$name`, `$uid`, and `$gid` items are returned. The `$dir` item contains the login directory in VMS syntax, while the `$comment` item contains the login directory in Unix syntax. The `$gcos` item contains the owner field from the UAF record. The `$quota` item is not used.

gmtime

The `gmtime` operator will function properly if you have a working CRTL `gmtime()` routine, or if the logical name SYS$TIMEZONE_DIFFERENTIAL is defined as the number of seconds which must be added to UTC to yield local time. (This logical name is defined automatically if you are running a version of VMS with built-in UTC support.) If neither of these cases is true, a warning message is printed, and `undef` is returned.

kill

In most cases, `kill` is implemented via the CRTL's `kill()` function, so it will behave according to that function's documentation. If you send a SIGKILL, however, the $DELPRC system service is called directly. This insures that the target process is actually deleted, if at all possible. (The CRTL's `kill()` function is presently implemented via $FORCEX, which is ignored by supervisor-mode images like DCL.)

Also, negative signal values don't do anything special under VMS; they're just converted to the corresponding positive value.

qx//

See the entry on `backticks` above.

select (system call)

If Perl was not built with socket support, the system call version of `select` is not available at all. If socket support is present, then the system call version of `select` functions only for file descriptors attached to sockets. It will not provide information about regular files or pipes, since the CRTL `select()` routine does not provide this functionality.

stat EXPR

Since VMS keeps track of files according to a different scheme than Unix, it's not really possible to represent the file's ID in the `st_dev` and `st_ino` fields of a `struct stat`. Perl tries its best, though, and the values it uses are pretty unlikely to be the same for two different

files. We can't guarantee this, though, so caveat scriptor.

system LIST

The `system` operator creates a subprocess, and passes its arguments to the subprocess for execution as a DCL command. Since the subprocess is created directly via `lib$spawn()`, any valid DCL command string may be specified. If the string begins with '@', it is treated as a DCL command unconditionally. Otherwise, if the first token contains a character used as a delimiter in file specification (e.g. `:` or `]`), an attempt is made to expand it using a default type of *.Exe* and the process defaults, and if successful, the resulting file is invoked via `MCR`. This allows you to invoke an image directly simply by passing the file specification to `system`, a common Unixish idiom. If the token has no file type, and matches a file with null type, then an attempt is made to determine whether the file is an executable image which should be invoked using `MCR` or a text file which should be passed to DCL as a command procedure.

If LIST consists of the empty string, `system` spawns an interactive DCL subprocess, in the same fashion as typing **SPAWN** at the DCL prompt.

Perl waits for the subprocess to complete before continuing execution in the current process. As described in *perlfunc*, the return value of `system` is a fake "status" which follows POSIX semantics unless the pragma `use vmsish 'status'` is in effect; see the description of `$?` in this document for more detail.

time

The value returned by `time` is the offset in seconds from 01-JAN-1970 00:00:00 (just like the CRTL's times() routine), in order to make life easier for code coming in from the POSIX/Unix world.

times

The array returned by the `times` operator is divided up according to the same rules the CRTL `times()` routine. Therefore, the "system time" elements will always be 0, since there is no difference between "user time" and "system" time under VMS, and the time accumulated by a subprocess may or may not appear separately in the "child time" field, depending on whether *times* keeps track of subprocesses separately. Note especially that the VAXCRTL (at least) keeps track only of subprocesses spawned using *fork* and *exec*; it will not accumulate the times of subprocesses spawned via pipes, *system*, or backticks.

unlink LIST

`unlink` will delete the highest version of a file only; in order to delete all versions, you need to say

```
1 while unlink LIST;
```

You may need to make this change to scripts written for a Unix system which expect that after a call to `unlink`, no files with the names passed to `unlink` will exist. (Note: This can be changed at compile time; if you `use Config` and `$Config{'d_unlink_all_versions'}` is `define`, then `unlink` will delete all versions of a file on the first call.)

`unlink` will delete a file if at all possible, even if it requires changing file protection (though it won't try to change the protection of the parent directory). You can tell whether you've got explicit delete access to a file by using the `VMS::Filespec::candelete` operator. For instance, in order to delete only files to which you have delete access, you could say something like

```
sub safe_unlink {
    my($file,$num);
    foreach $file (@_) {
        next unless VMS::Filespec::candelete($file);
        $num += unlink $file;
    }
```

```
        $num;
    }
```

(or you could just use `VMS::Stdio::remove`, if you've installed the VMS::Stdio extension distributed with Perl). If `unlink` has to change the file protection to delete the file, and you interrupt it in midstream, the file may be left intact, but with a changed ACL allowing you delete access.

This behavior of `unlink` is to be compatible with POSIX behavior and not traditional VMS behavior.

utime LIST

This operator changes only the modification time of the file (VMS revision date) on ODS-2 volumes and ODS-5 volumes without access dates enabled. On ODS-5 volumes with access dates enabled, the true access time is modified.

waitpid PID,FLAGS

If PID is a subprocess started by a piped `open()` (see *open*), `waitpid` will wait for that subprocess, and return its final status value in $?. If PID is a subprocess created in some other way (e.g. SPAWNed before Perl was invoked), `waitpid` will simply check once per second whether the process has completed, and return when it has. (If PID specifies a process that isn't a subprocess of the current process, and you invoked Perl with the `-w` switch, a warning will be issued.)

Returns PID on success, -1 on error. The FLAGS argument is ignored in all cases.

Perl variables

The following VMS-specific information applies to the indicated "special" Perl variables, in addition to the general information in *perlvar*. Where there is a conflict, this information takes precedence.

%ENV

The operation of the `%ENV` array depends on the translation of the logical name *PERL_ENV_TABLES*. If defined, it should be a search list, each element of which specifies a location for `%ENV` elements. If you tell Perl to read or set the element `$ENV{`*name*`}`, then Perl uses the translations of *PERL_ENV_TABLES* as follows:

CRTL_ENV

This string tells Perl to consult the CRTL's internal `environ` array of key-value pairs, using *name* as the key. In most cases, this contains only a few keys, but if Perl was invoked via the C `exec[lv]e()` function, as is the case for CGI processing by some HTTP servers, then the `environ` array may have been populated by the calling program.

CLISYM_[LOCAL]

A string beginning with `CLISYM_`tells Perl to consult the CLI's symbol tables, using *name* as the name of the symbol. When reading an element of `%ENV`, the local symbol table is scanned first, followed by the global symbol table.. The characters following `CLISYM_` are significant when an element of `%ENV` is set or deleted: if the complete string is `CLISYM_LOCAL`, the change is made in the local symbol table; otherwise the global symbol table is changed.

Any other string

If an element of *PERL_ENV_TABLES* translates to any other string, that string is used as the name of a logical name table, which is consulted using *name* as the logical name. The normal search order of access modes is used.

PERL_ENV_TABLES is translated once when Perl starts up; any changes you make while Perl is running do not affect the behavior of `%ENV`. If *PERL_ENV_TABLES* is not defined, then

Perl defaults to consulting first the logical name tables specified by *LNM$FILE_DEV*, and then the CRTL `environ` array.

In all operations on %ENV, the key string is treated as if it were entirely uppercase, regardless of the case actually specified in the Perl expression.

When an element of %ENV is read, the locations to which *PERL_ENV_TABLES* points are checked in order, and the value obtained from the first successful lookup is returned. If the name of the %ENV element contains a semi-colon, it and any characters after it are removed. These are ignored when the CRTL `environ` array or a CLI symbol table is consulted. However, the name is looked up in a logical name table, the suffix after the semi-colon is treated as the translation index to be used for the lookup. This lets you look up successive values for search list logical names. For instance, if you say

```
$  Define STORY  once,upon,a,time,there,was
$  perl -e "for ($i = 0; $i <= 6; $i++) " -
_$ -e "{ print $ENV{'story;'.$i},' '}"
```

Perl will print `ONCE UPON A TIME THERE WAS`, assuming, of course, that *PERL_ENV_TABLES* is set up so that the logical name `story` is found, rather than a CLI symbol or CRTL `environ` element with the same name.

When an element of %ENV is set to a defined string, the corresponding definition is made in the location to which the first translation of *PERL_ENV_TABLES* points. If this causes a logical name to be created, it is defined in supervisor mode. (The same is done if an existing logical name was defined in executive or kernel mode; an existing user or supervisor mode logical name is reset to the new value.) If the value is an empty string, the logical name's translation is defined as a single NUL (ASCII 00) character, since a logical name cannot translate to a zero-length string. (This restriction does not apply to CLI symbols or CRTL `environ` values; they are set to the empty string.) An element of the CRTL `environ` array can be set only if your copy of Perl knows about the CRTL's `setenv()` function. (This is present only in some versions of the DECCRTL; check `$Config{d_setenv}` to see whether your copy of Perl was built with a CRTL that has this function.)

When an element of %ENV is set to `undef`, the element is looked up as if it were being read, and if it is found, it is deleted. (An item "deleted" from the CRTL `environ` array is set to the empty string; this can only be done if your copy of Perl knows about the CRTL `setenv()` function.) Using `delete` to remove an element from %ENV has a similar effect, but after the element is deleted, another attempt is made to look up the element, so an inner-mode logical name or a name in another location will replace the logical name just deleted. In either case, only the first value found searching PERL_ENV_TABLES is altered. It is not possible at present to define a search list logical name via %ENV.

The element `$ENV{DEFAULT}` is special: when read, it returns Perl's current default device and directory, and when set, it resets them, regardless of the definition of *PERL_ENV_TABLES*. It cannot be cleared or deleted; attempts to do so are silently ignored.

Note that if you want to pass on any elements of the C-local environ array to a subprocess which isn't started by fork/exec, or isn't running a C program, you can "promote" them to logical names in the current process, which will then be inherited by all subprocesses, by saying

```
foreach my $key (qw[C-local keys you want promoted]) {
    my $temp = $ENV{$key}; # read from C-local array
    $ENV{$key} = $temp;    # and define as logical name
}
```

(You can't just say `$ENV{$key} = $ENV{$key}`, since the Perl optimizer is smart enough to elide the expression.)

Don't try to clear %ENV by saying `%ENV = ();`, it will throw a fatal error. This is equivalent to doing the following from DCL:

```
DELETE/LOGICAL *
```

You can imagine how bad things would be if, for example, the SYS$MANAGER or SYS$SYSTEM logical names were deleted.

At present, the first time you iterate over %ENV using `keys`, or `values`, you will incur a time penalty as all logical names are read, in order to fully populate %ENV. Subsequent iterations will not reread logical names, so they won't be as slow, but they also won't reflect any changes to logical name tables caused by other programs.

You do need to be careful with the logical names representing process-permanent files, such as `SYS$INPUT` and `SYS$OUTPUT`. The translations for these logical names are prepended with a two-byte binary value (0x1B 0x00) that needs to be stripped off if you wantto use it. (In previous versions of Perl it wasn't possible to get the values of these logical names, as the null byte acted as an end-of-string marker)

$!

The string value of `$!` is that returned by the CRTL's strerror() function, so it will include the VMS message for VMS-specific errors. The numeric value of `$!` is the value of `errno`, except if errno is EVMSERR, in which case `$!` contains the value of vaxc$errno. Setting `$!` always sets errno to the value specified. If this value is EVMSERR, it also sets vaxc$errno to 4 (NONAME-F-NOMSG), so that the string value of `$!` won't reflect the VMS error message from before `$!` was set.

$^E

This variable provides direct access to VMS status values in vaxc$errno, which are often more specific than the generic Unix-style error messages in `$!`. Its numeric value is the value of vaxc$errno, and its string value is the corresponding VMS message string, as retrieved by sys$getmsg(). Setting `$^E` sets vaxc$errno to the value specified.

While Perl attempts to keep the vaxc$errno value to be current, if errno is not EVMSERR, it may not be from the current operation.

$?

The "status value" returned in $? is synthesized from the actual exit status of the subprocess in a way that approximates POSIX wait(5) semantics, in order to allow Perl programs to portably test for successful completion of subprocesses. The low order 8 bits of $? are always 0 under VMS, since the termination status of a process may or may not have been generated by an exception.

The next 8 bits contain the termination status of the program.

If the child process follows the convention of C programs compiled with the _POSIX_EXIT macro set, the status value will contain the actual value of 0 to 255 returned by that program on a normal exit.

With the _POSIX_EXIT macro set, the UNIX exit value of zero is represented as a VMS native status of 1, and the UNIX values from 2 to 255 are encoded by the equation:

```
VMS_status = 0x35a000 + (unix_value * 8) + 1.
```

And in the special case of unix value 1 the encoding is:

```
VMS_status = 0x35a000 + 8 + 2 + 0x10000000.
```

For other termination statuses, the severity portion of the subprocess' exit status is used: if the severity was success or informational, these bits are all 0; if the severity was warning, they contain a value of 1; if the severity was error or fatal error, they contain the actual severity bits, which turns out to be a value of 2 for error and 4 for severe_error. Fatal is another term for the severe_error status.

As a result, $? will always be zero if the subprocess' exit status indicated successful

completion, and non-zero if a warning or error occurred or a program compliant with encoding _POSIX_EXIT values was run and set a status.

How can you tell the difference between a non-zero status that is the result of a VMS native error status or an encoded UNIX status? You can not unless you look at the ${^CHILD_ERROR_NATIVE} value. The ${^CHILD_ERROR_NATIVE} value returns the actual VMS status value and check the severity bits. If the severity bits are equal to 1, then if the numeric value for $? is between 2 and 255 or 0, then $? accurately reflects a value passed back from a UNIX application. If $? is 1, and the severity bits indicate a VMS error (2), then $? is from a UNIX application exit value.

In practice, Perl scripts that call programs that return _POSIX_EXIT type status values will be expecting those values, and programs that call traditional VMS programs will either be expecting the previous behavior or just checking for a non-zero status.

And success is always the value 0 in all behaviors.

When the actual VMS termination status of the child is an error, internally the `$!` value will be set to the closest UNIX errno value to that error so that Perl scripts that test for error messages will see the expected UNIX style error message instead of a VMS message.

Conversely, when setting $? in an END block, an attempt is made to convert the POSIX value into a native status intelligible to the operating system upon exiting Perl. What this boils down to is that setting $? to zero results in the generic success value SS$_NORMAL, and setting $? to a non-zero value results in the generic failure status SS$_ABORT. See also *"exit" in perlport*.

With the future POSIX_EXIT mode set, setting $? will cause the new value to also be encoded into $^E so that the either the original parent or child exit status values of 0 to 255 can be automatically recovered by C programs expecting _POSIX_EXIT behavior. If both a parent and a child exit value are non-zero, then it will be assumed that this is actually a VMS native status value to be passed through. The special value of 0xFFFF is almost a NOOP as it will cause the current native VMS status in the C library to become the current native Perl VMS status, and is handled this way as consequence of it known to not be a valid native VMS status value. It is recommend that only values in range of normal UNIX parent or child status numbers, 0 to 255 are used.

The pragma `use vmsish 'status'` makes $? reflect the actual VMS exit status instead of the default emulation of POSIX status described above. This pragma also disables the conversion of non-zero values to SS$_ABORT when setting $? in an END block (but zero will still be converted to SS$_NORMAL).

Do not use the pragma `use vmsish 'status'` with the future POSIX_EXIT mode, as they are at times requesting conflicting actions and the consequence of ignoring this advice will be undefined to allow future improvements in the POSIX exit handling.

$|

Setting `$|` for an I/O stream causes data to be flushed all the way to disk on each write (*i.e.* not just to the underlying RMS buffers for a file). In other words, it's equivalent to calling fflush() and fsync() from C.

Standard modules with VMS-specific differences

SDBM_File

SDBM_File works properly on VMS. It has, however, one minor difference. The database directory file created has a *.sdbm_dir* extension rather than a *.dir* extension. *.dir* files are VMS filesystem directory files, and using them for other purposes could cause unacceptable problems.

Revision date

This document was last updated on 14-Oct-2005, for Perl 5, patchlevel 8.

AUTHOR

Charles Bailey bailey@cor.newman.upenn.edu Craig Berry craigberry@mac.com Dan Sugalski dan@sidhe.org John Malmberg wb8tyw@qsl.net