NAME

perlwin32 - Perl under Windows

SYNOPSIS

These are instructions for building Perl under Windows 9x/NT/2000/XP on the Intel x86 and Itanium architectures.

DESCRIPTION

Before you start, you should glance through the README file found in the top-level directory to which the Perl distribution was extracted. Make sure you read and understand the terms under which this software is being distributed.

Also make sure you read *BUGS AND CAVEATS* below for the known limitations of this port.

The INSTALL file in the perl top-level has much information that is only relevant to people building Perl on Unix-like systems. In particular, you can safely ignore any information that talks about "Configure".

You may also want to look at two other options for building a perl that will work on Windows NT: the README.cygwin and README.os2 files, each of which give a different set of rules to build a Perl that will work on Win32 platforms. Those two methods will probably enable you to build a more Unix-compatible perl, but you will also need to download and use various other build-time and run-time support software described in those files.

This set of instructions is meant to describe a so-called "native" port of Perl to Win32 platforms. This includes both 32-bit and 64-bit Windows operating systems. The resulting Perl requires no additional software to run (other than what came with your operating system). Currently, this port is capable of using one of the following compilers on the Intel x86 architecture:

```
Borland C++           version 5.02 or later
Microsoft Visual C++  version 2.0 or later
MinGW with gcc        gcc version 2.95.2 or later
```

The last of these is a high quality freeware compiler. Use version 3.2.x or later for the best results with this compiler.

The Borland C++ and Microsoft Visual C++ compilers are also now being given away free. The Borland compiler is available as "Borland C++ Compiler Free Command Line Tools" and is the same compiler that ships with the full "Borland C++ Builder" product. The Microsoft compiler is available as "Visual C++ Toolkit 2003", and also as part of the ".NET Framework SDK", and is the same compiler that ships with "Visual Studio .NET 2003 Professional".

This port can also be built on the Intel IA64 using:

```
Microsoft Platform SDK Nov 2001 (64-bit compiler and tools)
```

The MS Platform SDK can be downloaded from http://www.microsoft.com/.

This port fully supports MakeMaker (the set of modules that is used to build extensions to perl). Therefore, you should be able to build and install most extensions found in the CPAN sites. See *Usage Hints for Perl on Win32* below for general hints about this.

Setting Up Perl on Win32

Make

You need a "make" program to build the sources. If you are using Visual C++ or the Platform SDK tools under Windows NT/2000/XP, nmake will work. All other builds need dmake.

dmake is a freely available make that has very nice macro features and parallelability.

A port of dmake for Windows is available from:

```
http://search.cpan.org/dist/dmake/
```

Fetch and install dmake somewhere on your path.

There exists a minor coexistence problem with dmake and Borland C++ compilers. Namely, if a distribution has C files named with mixed case letters, they will be compiled into appropriate .obj-files named with all lowercase letters, and every time dmake is invoked to bring files up to date, it will try to recompile such files again. For example, Tk distribution has a lot of such files, resulting in needless recompiles every time dmake is invoked. To avoid this, you may use the script "sync_ext.pl" after a successful build. It is available in the win32 subdirectory of the Perl source distribution.

Command Shell

Use the default "cmd" shell that comes with NT. Some versions of the popular 4DOS/NT shell have incompatibilities that may cause you trouble. If the build fails under that shell, try building again with the cmd shell.

The nmake Makefile also has known incompatibilities with the "command.com" shell that comes with Windows 9x. You will need to use dmake and makefile.mk to build under Windows 9x.

The surest way to build it is on Windows NT/2000/XP, using the cmd shell.

Make sure the path to the build directory does not contain spaces. The build usually works in this circumstance, but some tests will fail.

Borland C++

If you are using the Borland compiler, you will need dmake. (The make that Borland supplies is seriously crippled and will not work for MakeMaker builds.)

See *Make* above.

Microsoft Visual C++

The nmake that comes with Visual C++ will suffice for building. You will need to run the VCVARS32.BAT file, usually found somewhere like C:\MSDEV4.2\BIN or C:\Program Files\Microsoft Visual Studio\VC98\Bin. This will set your build environment.

You can also use dmake to build using Visual C++; provided, however, you set OSRELEASE to "microsft" (or whatever the directory name under which the Visual C dmake configuration lives) in your environment and edit win32/config.vc to change "make=nmake" into "make=dmake". The latter step is only essential if you want to use dmake as your default make for building extensions using MakeMaker.

Microsoft Visual C++ Toolkit 2003

This free toolkit contains the same compiler and linker that ship with Visual Studio .NET 2003 Professional, but doesn't contain everything necessary to build Perl.

You will also need to download the "Platform SDK" (the "Core SDK" and "MDAC SDK" components are required) for header files, libraries and rc.exe, and ".NET Framework SDK" for more libraries and nmake.exe. Note that the latter (which also includes the free compiler and linker) requires the ".NET Framework Redistributable" to be installed first. This can be downloaded and installed separately, but is included in the "Visual C++ Toolkit 2003" anyway.

These packages can all be downloaded by searching in the Download Center at http://www.microsoft.com/downloads/search.aspx?displaylang=en. (Providing exact links to these packages has proven a pointless task because the links keep on changing so often.)

Try to obtain the latest version of the Platform SDK. Sometimes these packages contain a particular Windows OS version in their name, but actually work on other OS versions too. For example, the "Windows Server 2003 SP1 Platform SDK" also runs on Windows XP SP2 and Windows 2000.

According to the download pages the Toolkit and the .NET Framework SDK are only supported on Windows 2000/XP/2003, so trying to use these tools on Windows 95/98/ME and even Windows NT probably won't work.

Install the Toolkit first, then the Platform SDK, then the .NET Framework SDK. Setup your environment as follows (assuming default installation locations were chosen):

```
 SET PATH=%SystemRoot%\system32;%SystemRoot%;C:\Program
Files\Microsoft Visual C++ Toolkit 2003\bin;C:\Program
Files\Microsoft SDK\Bin;C:\Program Files\Microsoft.NET\SDK\v1.1\Bin
 SET INCLUDE=C:\Program Files\Microsoft Visual C++ Toolkit
2003\include;C:\Program Files\Microsoft SDK\include;C:\Program
Files\Microsoft Visual Studio .NET 2003\Vc7\include
 SET LIB=C:\Program Files\Microsoft Visual C++ Toolkit
2003\lib;C:\Program Files\Microsoft SDK\lib;C:\Program
Files\Microsoft Visual Studio .NET 2003\Vc7\lib
```

Several required files will still be missing:

- cvtres.exe is required by link.exe when using a .res file. It is actually installed by the .NET Framework SDK, but into a location such as the following:

  ```
  C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322
  ```

  Copy it from there to C:\Program Files\Microsoft SDK\Bin

- lib.exe is normally used to build libraries, but link.exe with the /lib option also works, so change win32/config.vc to use it instead:

  Change the line reading:

  ```
  ar='lib'
  ```

  to:

  ```
  ar='link /lib'
  ```

  It may also be useful to create a batch file called lib.bat in C:\Program Files\Microsoft Visual C++ Toolkit 2003\bin containing:

  ```
  @echo off
  link /lib %*
  ```

  for the benefit of any naughty C extension modules that you might want to build later which explicitly reference "lib" rather than taking their value from $Config{ar}.

- setargv.obj is required to build perlglob.exe (and perl.exe if the USE_SETARGV option is enabled). The Platform SDK supplies this object file in source form in C:\Program Files\Microsoft SDK\src\crt. Copy setargv.c, cruntime.h and internal.h from there to some temporary location and build setargv.obj using

  ```
  cl.exe /c /I. /D_CRTBLD setargv.c
  ```

  Then copy setargv.obj to C:\Program Files\Microsoft SDK\lib

  Alternatively, if you don't need perlglob.exe and don't need to enable the USE_SETARGV option then you can safely just remove all mention of $(GLOBEXE) from win32/Makefile and setargv.obj won't be required anyway.

Perl should now build using the win32/Makefile. You will need to edit that file to set

```
 CCTYPE = MSVC70FREE
```

and to set CCHOME, CCINCDIR and CCLIBDIR as per the environment setup above.

Microsoft Platform SDK 64-bit Compiler

The nmake that comes with the Platform SDK will suffice for building Perl. Make sure you are building within one of the "Build Environment" shells available after you install the Platform SDK from the Start Menu.

MinGW release 3 with gcc

The latest release of MinGW at the time of writing is 3.1.0, which contains gcc-3.2.3. It can be downloaded here:

```
http://www.mingw.org/
```

Perl also compiles with earlier releases of gcc (2.95.2 and up). See below for notes about using earlier versions of MinGW/gcc.

You also need dmake. See *Make* above on how to get it.

MinGW release 1 with gcc

The MinGW-1.1 bundle contains gcc-2.95.3.

Make sure you install the binaries that work with MSVCRT.DLL as indicated in the README for the GCC bundle. You may need to set up a few environment variables (usually ran from a batch file).

There are a couple of problems with the version of gcc-2.95.2-msvcrt.exe released 7 November 1999:

- It left out a fix for certain command line quotes. To fix this, be sure to download and install the file fixes/quote-fix-msvcrt.exe from the above ftp location.
- The definition of the fpos_t type in stdio.h may be wrong. If your stdio.h has this problem, you will see an exception when running the test t/lib/io_xs.t. To fix this, change the typedef for fpos_t from "long" to "long long" in the file i386-mingw32msvc/include/stdio.h, and rebuild.

A potentially simpler to install (but probably soon-to-be-outdated) bundle of the above package with the mentioned fixes already applied is available here:

```
http://downloads.ActiveState.com/pub/staff/gsar/gcc-2.95.2-msvcrt.zip
    ftp://ftp.ActiveState.com/pub/staff/gsar/gcc-2.95.2-msvcrt.zip
```

Building

- Make sure you are in the "win32" subdirectory under the perl toplevel. This directory contains a "Makefile" that will work with versions of nmake that come with Visual C++ or the Platform SDK, and a dmake "makefile.mk" that will work for all supported compilers. The defaults in the dmake makefile are setup to build using MinGW/gcc.
- Edit the makefile.mk (or Makefile, if you're using nmake) and change the values of INST_DRV and INST_TOP. You can also enable various build flags. These are explained in the makefiles.

  Note that it is generally not a good idea to try to build a perl with INST_DRV and INST_TOP set to a path that already exists from a previous build. In particular, this may cause problems with the lib/ExtUtils/t/Embed.t test, which attempts to build a test program and may end up building against the installed perl's lib/CORE directory rather than the one being tested.

  You will have to make sure that CCTYPE is set correctly and that CCHOME points to wherever you installed your compiler.

  The default value for CCHOME in the makefiles for Visual C++ may not be correct for some versions. Make sure the default exists and is valid.

  You may also need to comment out the `DELAYLOAD = ...` line in the Makefile if you're using

VC++ 6.0 without the latest service pack and the linker reports an internal error.

If you have either the source or a library that contains des_fcrypt(), enable the appropriate option in the makefile. A ready-to-use version of fcrypt.c, based on the version originally written by Eric Young at ftp://ftp.funet.fi/pub/crypt/mirrors/dsi/libdes/, is bundled with the distribution and CRYPT_SRC is set to use it. Alternatively, if you have built a library that contains des_fcrypt(), you can set CRYPT_LIB to point to the library name. Perl will also build without des_fcrypt(), but the crypt() builtin will fail at run time.

If you want build some core extensions statically into perl's dll, specify them in the STATIC_EXT macro.

Be sure to read the instructions near the top of the makefiles carefully.

- Type "dmake" (or "nmake" if you are using that make).

  This should build everything. Specifically, it will create perl.exe, perl58.dll at the perl toplevel, and various other extension dll's under the lib\auto directory. If the build fails for any reason, make sure you have done the previous steps correctly.

Testing Perl on Win32

Type "dmake test" (or "nmake test"). This will run most of the tests from the testsuite (many tests will be skipped).

There should be no test failures when running under Windows NT/2000/XP. Many tests *will* fail under Windows 9x due to the inferior command shell.

Some test failures may occur if you use a command shell other than the native "cmd.exe", or if you are building from a path that contains spaces. So don't do that.

If you are running the tests from a emacs shell window, you may see failures in op/stat.t. Run "dmake test-notty" in that case.

If you're using the Borland compiler, you may see a failure in op/taint.t arising from the inability to find the Borland Runtime DLLs on the system default path. You will need to copy the DLLs reported by the messages from where Borland chose to install it, into the Windows system directory (usually somewhere like C:\WINNT\SYSTEM32) and rerun the test.

If you're using Borland compiler versions 5.2 and below, you may run into problems finding the correct header files when building extensions. For example, building the "Tk" extension may fail because both perl and Tk contain a header file called "patchlevel.h". The latest Borland compiler (v5.5) is free of this misbehaviour, and it even supports an option -VI- for backward (bugward) compatibility for using the old Borland search algorithm to locate header files.

If you run the tests on a FAT partition, you may see some failures for `link()` related tests:

```
    Failed Test                    Stat Wstat Total Fail  Failed  List

    ../ext/IO/lib/IO/t/io_dup.t                6    4  66.67%  2-5
    ../lib/File/Temp/t/mktemp.t                9    1  11.11%  2
    ../lib/File/Temp/t/posix.t                 7    1  14.29%  3
    ../lib/File/Temp/t/security.t             13    1   7.69%  2
    ../lib/File/Temp/t/tempfile.t             20    2  10.00%  2 4
    comp/multiline.t                           6    2  33.33%  5-6
    io/dup.t                                   8    6  75.00%  2-7
    op/write.t                                47    7  14.89%  1-3 6
9-11
```

Testing on NTFS avoids these errors.

Furthermore, you should make sure that during `make test` you do not have any GNU tool packages

in your path: some toolkits like Unixutils include some tools (`type` for instance) which override the Windows ones and makes tests fail. Remove them from your path while testing to avoid these errors.

Please report any other failures as described under *BUGS AND CAVEATS*.

Installation of Perl on Win32

Type "dmake install" (or "nmake install"). This will put the newly built perl and the libraries under whatever `INST_TOP` points to in the Makefile. It will also install the pod documentation under `$INST_TOP\$INST_VER\lib\pod` and HTML versions of the same under `$INST_TOP\$INST_VER\lib\pod\html`.

To use the Perl you just installed you will need to add a new entry to your PATH environment variable: `$INST_TOP\bin`, e.g.

```
set PATH=c:\perl\bin;%PATH%
```

If you opted to uncomment `INST_VER` and `INST_ARCH` in the makefile then the installation structure is a little more complicated and you will need to add two new PATH components instead: `$INST_TOP\$INST_VER\bin` and `$INST_TOP\$INST_VER\bin\$ARCHNAME`, e.g.

```
set PATH=c:\perl\5.6.0\bin;c:\perl\5.6.0\bin\MSWin32-x86;%PATH%
```

Usage Hints for Perl on Win32

Environment Variables

The installation paths that you set during the build get compiled into perl, so you don't have to do anything additional to start using that perl (except add its location to your PATH variable).

If you put extensions in unusual places, you can set PERL5LIB to a list of paths separated by semicolons where you want perl to look for libraries. Look for descriptions of other environment variables you can set in *perlrun*.

You can also control the shell that perl uses to run system() and backtick commands via PERL5SHELL. See *perlrun*.

Perl does not depend on the registry, but it can look up certain default values if you choose to put them there. Perl attempts to read entries from `HKEY_CURRENT_USER\Software\Perl` and `HKEY_LOCAL_MACHINE\Software\Perl`. Entries in the former override entries in the latter. One or more of the following entries (of type REG_SZ or REG_EXPAND_SZ) may be set:

```
lib-$]  version-specific standard library path to add to @INC
lib   standard library path to add to @INC
sitelib-$]  version-specific site library path to add to @INC
sitelib  site library path to add to @INC
vendorlib-$] version-specific vendor library path to add to @INC
vendorlib  vendor library path to add to @INC
PERL*  fallback for all %ENV lookups that begin with "PERL"
```

Note the `$]` in the above is not literal. Substitute whatever version of perl you want to honor that entry, e.g. `5.6.0`. Paths must be separated with semicolons, as usual on win32.

File Globbing

By default, perl handles file globbing using the File::Glob extension, which provides portable globbing.

If you want perl to use globbing that emulates the quirks of DOS filename conventions, you might want to consider using File::DosGlob to override the internal glob() implementation. See *File::DosGlob* for details.

Using perl from the command line

If you are accustomed to using perl from various command-line shells found in UNIX environments, you will be less than pleased with what Windows offers by way of a command shell.

The crucial thing to understand about the Windows environment is that the command line you type in is processed twice before Perl sees it. First, your command shell (usually CMD.EXE on Windows NT, and COMMAND.COM on Windows 9x) preprocesses the command line, to handle redirection, environment variable expansion, and location of the executable to run. Then, the perl executable splits the remaining command line into individual arguments, using the C runtime library upon which Perl was built.

It is particularly important to note that neither the shell nor the C runtime do any wildcard expansions of command-line arguments (so wildcards need not be quoted). Also, the quoting behaviours of the shell and the C runtime are rudimentary at best (and may, if you are using a non-standard shell, be inconsistent). The only (useful) quote character is the double quote ("). It can be used to protect spaces and other special characters in arguments.

The Windows NT documentation has almost no description of how the quoting rules are implemented, but here are some general observations based on experiments: The C runtime breaks arguments at spaces and passes them to programs in argc/argv. Double quotes can be used to prevent arguments with spaces in them from being split up. You can put a double quote in an argument by escaping it with a backslash and enclosing the whole argument within double quotes. The backslash and the pair of double quotes surrounding the argument will be stripped by the C runtime.

The file redirection characters "<", ">", and "|" can be quoted by double quotes (although there are suggestions that this may not always be true). Single quotes are not treated as quotes by the shell or the C runtime, they don't get stripped by the shell (just to make this type of quoting completely useless). The caret "^" has also been observed to behave as a quoting character, but this appears to be a shell feature, and the caret is not stripped from the command line, so Perl still sees it (and the C runtime phase does not treat the caret as a quote character).

Here are some examples of usage of the "cmd" shell:

This prints two doublequotes:

```
perl -e "print '\"\"' "
```

This does the same:

```
perl -e "print \"\\\"\\\"\" "
```

This prints "bar" and writes "foo" to the file "blurch":

```
perl -e "print 'foo'; print STDERR 'bar'" > blurch
```

This prints "foo" ("bar" disappears into nowhereland):

```
perl -e "print 'foo'; print STDERR 'bar'" 2> nul
```

This prints "bar" and writes "foo" into the file "blurch":

```
perl -e "print 'foo'; print STDERR 'bar'" 1> blurch
```

This pipes "foo" to the "less" pager and prints "bar" on the console:

```
perl -e "print 'foo'; print STDERR 'bar'" | less
```

This pipes "foo\nbar\n" to the less pager:

```
perl -le "print 'foo'; print STDERR 'bar'" 2>&1 | less
```

This pipes "foo" to the pager and writes "bar" in the file "blurch":

```
perl -e "print 'foo'; print STDERR 'bar'" 2> blurch | less
```

Discovering the usefulness of the "command.com" shell on Windows 9x is left as an exercise to the reader :)

One particularly pernicious problem with the 4NT command shell for Windows NT is that it (nearly) always treats a % character as indicating that environment variable expansion is needed. Under this shell, it is therefore important to always double any % characters which you want Perl to see (for example, for hash variables), even when they are quoted.

Building Extensions

The Comprehensive Perl Archive Network (CPAN) offers a wealth of extensions, some of which require a C compiler to build. Look in http://www.cpan.org/ for more information on CPAN.

Note that not all of the extensions available from CPAN may work in the Win32 environment; you should check the information at http://testers.cpan.org/ before investing too much effort into porting modules that don't readily build.

Most extensions (whether they require a C compiler or not) can be built, tested and installed with the standard mantra:

```
perl Makefile.PL
$MAKE
$MAKE test
$MAKE install
```

where $MAKE is whatever 'make' program you have configured perl to use. Use "perl -V:make" to find out what this is. Some extensions may not provide a testsuite (so "$MAKE test" may not do anything or fail), but most serious ones do.

It is important that you use a supported 'make' program, and ensure Config.pm knows about it. If you don't have nmake, you can either get dmake from the location mentioned earlier or get an old version of nmake reportedly available from:

```
http://download.microsoft.com/download/vc15/Patch/1.52/W95/EN-US/nmake15.exe
```

Another option is to use the make written in Perl, available from CPAN.

```
http://www.cpan.org/modules/by-module/Make/
```

You may also use dmake. See *Make* above on how to get it.

Note that MakeMaker actually emits makefiles with different syntax depending on what 'make' it thinks you are using. Therefore, it is important that one of the following values appears in Config.pm:

```
make='nmake' # MakeMaker emits nmake syntax
make='dmake' # MakeMaker emits dmake syntax
any other value # MakeMaker emits generic make syntax
       (e.g GNU make, or Perl make)
```

If the value doesn't match the 'make' program you want to use, edit Config.pm to fix it.

If a module implements XSUBs, you will need one of the supported C compilers. You must make sure you have set up the environment for the compiler for command-line compilation.

If a module does not build for some reason, look carefully for why it failed, and report problems to the module author. If it looks like the extension building support is at fault, report that with full details of how the build failed using the perlbug utility.

Command-line Wildcard Expansion

The default command shells on DOS descendant operating systems (such as they are)

usually do not expand wildcard arguments supplied to programs. They consider it the application's job to handle that. This is commonly achieved by linking the application (in our case, perl) with startup code that the C runtime libraries usually provide. However, doing that results in incompatible perl versions (since the behavior of the argv expansion code differs depending on the compiler, and it is even buggy on some compilers). Besides, it may be a source of frustration if you use such a perl binary with an alternate shell that *does* expand wildcards.

Instead, the following solution works rather well. The nice things about it are 1) you can start using it right away; 2) it is more powerful, because it will do the right thing with a pattern like */*/*.c; 3) you can decide whether you do/don't want to use it; and 4) you can extend the method to add any customizations (or even entirely different kinds of wildcard expansion).

```
C:\> copy con c:\perl\lib\Wild.pm
# Wild.pm - emulate shell @ARGV expansion on shells that don't
use File::DosGlob;
@ARGV = map {
       my @g = File::DosGlob::glob($_) if /[*?]/;
       @g ? @g : $_;
     } @ARGV;
1;
^Z
C:\> set PERL5OPT=-MWild
C:\> perl -le "for (@ARGV) { print }" */*/perl*.c
p4view/perl/perl.c
p4view/perl/perlio.c
p4view/perl/perly.c
perl5.005/win32/perlglob.c
perl5.005/win32/perllib.c
perl5.005/win32/perlglob.c
perl5.005/win32/perllib.c
perl5.005/win32/perlglob.c
perl5.005/win32/perllib.c
```

Note there are two distinct steps there: 1) You'll have to create Wild.pm and put it in your perl lib directory. 2) You'll need to set the PERL5OPT environment variable. If you want argv expansion to be the default, just set PERL5OPT in your default startup environment.

If you are using the Visual C compiler, you can get the C runtime's command line wildcard expansion built into perl binary. The resulting binary will always expand unquoted command lines, which may not be what you want if you use a shell that does that for you. The expansion done is also somewhat less powerful than the approach suggested above.

Win32 Specific Extensions

A number of extensions specific to the Win32 platform are available from CPAN. You may find that many of these extensions are meant to be used under the Activeware port of Perl, which used to be the only native port for the Win32 platform. Since the Activeware port does not have adequate support for Perl's extension building tools, these extensions typically do not support those tools either and, therefore, cannot be built using the generic steps shown in the previous section.

To ensure smooth transitioning of existing code that uses the ActiveState port, there is a bundle of Win32 extensions that contains all of the ActiveState extensions and several other Win32 extensions from CPAN in source form, along with many added bugfixes, and with MakeMaker support. The latest version of this bundle is available at:

```
http://search.cpan.org/dist/libwin32/
```

See the README in that distribution for building and installation instructions.

Notes on 64-bit Windows

Windows .NET Server supports the LLP64 data model on the Intel Itanium architecture.

The LLP64 data model is different from the LP64 data model that is the norm on 64-bit Unix platforms. In the former, `int` and `long` are both 32-bit data types, while pointers are 64 bits wide. In addition, there is a separate 64-bit wide integral type, `__int64`. In contrast, the LP64 data model that is pervasive on Unix platforms provides `int` as the 32-bit type, while both the `long` type and pointers are of 64-bit precision. Note that both models provide for 64-bits of addressability.

64-bit Windows running on Itanium is capable of running 32-bit x86 binaries transparently. This means that you could use a 32-bit build of Perl on a 64-bit system. Given this, why would one want to build a 64-bit build of Perl? Here are some reasons why you would bother:

- A 64-bit native application will run much more efficiently on Itanium hardware.
- There is no 2GB limit on process size.
- Perl automatically provides large file support when built under 64-bit Windows.
- Embedding Perl inside a 64-bit application.

Running Perl Scripts

Perl scripts on UNIX use the "#!" (a.k.a "shebang") line to indicate to the OS that it should execute the file using perl. Win32 has no comparable means to indicate arbitrary files are executables.

Instead, all available methods to execute plain text files on Win32 rely on the file "extension". There are three methods to use this to execute perl scripts:

1 There is a facility called "file extension associations" that will work in Windows NT 4.0. This can be manipulated via the two commands "assoc" and "ftype" that come standard with Windows NT 4.0. Type "ftype /?" for a complete example of how to set this up for perl scripts (Say what? You thought Windows NT wasn't perl-ready? :).

2 Since file associations don't work everywhere, and there are reportedly bugs with file associations where it does work, the old method of wrapping the perl script to make it look like a regular batch file to the OS, may be used. The install process makes available the "pl2bat.bat" script which can be used to wrap perl scripts into batch files. For example:

```
pl2bat foo.pl
```

will create the file "FOO.BAT". Note "pl2bat" strips any .pl suffix and adds a .bat suffix to the generated file.

If you use the 4DOS/NT or similar command shell, note that "pl2bat" uses the "%*" variable in the generated batch file to refer to all the command line arguments, so you may need to make sure that construct works in batch files. As of this writing, 4DOS/NT users will need a "ParameterChar = *" statement in their 4NT.INI file or will need to execute "setdos /p*" in the 4DOS/NT startup file to enable this to work.

3 Using "pl2bat" has a few problems: the file name gets changed, so scripts that rely on `$0` to find what they must do may not run properly; running "pl2bat" replicates the contents of the original script, and so this process can be maintenance intensive if the originals get updated often. A different approach that avoids both problems is possible.

A script called "runperl.bat" is available that can be copied to any filename (along with the .bat suffix). For example, if you call it "foo.bat", it will run the file "foo" when it is executed. Since you can run batch files on Win32 platforms simply by typing the name (without the extension), this effectively runs the file "foo", when you type either "foo" or "foo.bat". With this method, "foo.bat" can even be in a different location than the file "foo", as long as "foo" is available somewhere on the PATH. If your scripts are on a

filesystem that allows symbolic links, you can even avoid copying "runperl.bat".

Here's a diversion: copy "runperl.bat" to "runperl", and type "runperl". Explain the observed behavior, or lack thereof. :) Hint: .gnidnats llits er'uoy fi ,"lrepnur" eteled :tniH

Miscellaneous Things

A full set of HTML documentation is installed, so you should be able to use it if you have a web browser installed on your system.

`perldoc` is also a useful tool for browsing information contained in the documentation, especially in conjunction with a pager like `less` (recent versions of which have Win32 support). You may have to set the PAGER environment variable to use a specific pager. "perldoc -f foo" will print information about the perl operator "foo".

One common mistake when using this port with a GUI library like `Tk` is assuming that Perl's normal behavior of opening a command-line window will go away. This isn't the case. If you want to start a copy of `perl` without opening a command-line window, use the `wperl` executable built during the installation process. Usage is exactly the same as normal `perl` on Win32, except that options like `-h` don't work (since they need a command-line window to print to).

If you find bugs in perl, you can run `perlbug` to create a bug report (you may have to send it manually if `perlbug` cannot find a mailer on your system).

BUGS AND CAVEATS

Norton AntiVirus interferes with the build process, particularly if set to "AutoProtect, All Files, when Opened". Unlike large applications the perl build process opens and modifies a lot of files. Having the the AntiVirus scan each and every one slows build the process significantly. Worse, with PERLIO=stdio the build process fails with peculiar messages as the virus checker interacts badly with miniperl.exe writing configure files (it seems to either catch file part written and treat it as suspicious, or virus checker may have it "locked" in a way which inhibits miniperl updating it). The build does complete with

```
set PERLIO=perlio
```

but that may be just luck. Other AntiVirus software may have similar issues.

Some of the built-in functions do not act exactly as documented in *perlfunc*, and a few are not implemented at all. To avoid surprises, particularly if you have had prior exposure to Perl in other operating environments or if you intend to write code that will be portable to other environments, see *perlport* for a reasonably definitive list of these differences.

Not all extensions available from CPAN may build or work properly in the Win32 environment. See *Building Extensions*.

Most `socket()` related calls are supported, but they may not behave as on Unix platforms. See *perlport* for the full list. Perl requires Winsock2 to be installed on the system. If you're running Win95, you can download Winsock upgrade from here:

http://www.microsoft.com/windows95/downloads/contents/WUAdminTools/S_WUNetworkingTools/W95Sockets2/Default.asp

Later OS versions already include Winsock2 support.

Signal handling may not behave as on Unix platforms (where it doesn't exactly "behave", either :). For instance, calling `die()` or `exit()` from signal handlers will cause an exception, since most implementations of `signal()` on Win32 are severely crippled. Thus, signals may work only for simple things like setting a flag variable in the handler. Using signals under this port should currently be considered unsupported.

Please send detailed descriptions of any problems and solutions that you may find to <

perlbug@perl.org>, along with the output produced by `perl -V`.

ACKNOWLEDGEMENTS

AUTHORS

Gary Ng <71564.1743@CompuServe.COM>

Gurusamy Sarathy <gsar@activestate.com>

Nick Ing-Simmons <nick@ing-simmons.net>

Jan Dubois <jand@activestate.com>

Steve Hay <steve.hay@uk.radan.com>

This document is maintained by Jan Dubois.

SEE ALSO

perl

HISTORY

This port was originally contributed by Gary Ng around 5.003_24, and borrowed from the Hip Communications port that was available at the time. Various people have made numerous and sundry hacks since then.

Borland support was added in 5.004_01 (Gurusamy Sarathy).

GCC/mingw32 support was added in 5.005 (Nick Ing-Simmons).

Support for PERL_OBJECT was added in 5.005 (ActiveState Tool Corp).

Support for fork() emulation was added in 5.6 (ActiveState Tool Corp).

Win9x support was added in 5.6 (Benjamin Stuhl).

Support for 64-bit Windows added in 5.8 (ActiveState Corp).

Last updated: 30 September 2005