PCRE LICENCE

PCRE is a library of functions to support regular expressions whose and semantics are as close as possible to those of the Perl 5 langua

Release 7 of PCRE is distributed under the terms of the "BSD" licenc specified below. The documentation for PCRE, supplied in the "doc" directory, is distributed under the same terms as the software itsel

The basic library functions are written in C and are freestanding. A included in the distribution is a set of C++ wrapper functions.

THE BASIC LIBRARY FUNCTIONS

Written by: Philip Hazel Email local part: ph10 Email domain: cam.ac.uk

University of Cambridge Computing Service, Cambridge, England.

Copyright (c) 1997-2007 University of Cambridge All rights reserved.

THE C++ WRAPPER FUNCTIONS

Contributed by: Google Inc.

Copyright (c) 2007, Google Inc. All rights reserved.

THE "BSD" LICENCE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions a

- * Redistributions of source code must retain the above copyright this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyri notice, this list of conditions and the following disclaimer i documentation and/or other materials provided with the distrib

* Neither the name of the University of Cambridge nor the name c Inc. nor the names of their contributors may be used to endors promote products derived from this software without specific p written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR P ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTOR LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSI INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWI ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED POSSIBILITY OF SUCH DAMAGE.

End

THE MAIN PCRE LIBRARY

Written by: Philip Hazel Email local part: ph10 Email domain: cam.ac.uk

University of Cambridge Computing Service, Cambridge, England.

Copyright (c) 1997-2007 University of Cambridge All rights reserved

THE C++ WRAPPER LIBRARY

Written by: Google Inc.

Copyright (c) 2007 Google Inc All rights reserved

####

Windows CHM file contributed by Sheri Pierce

News about PCRE releases

Release 7.2 19-Jun-07

WARNING: saved patterns that were compiled by earlier versions of PC recompiled for use with 7.2 (necessitated by the addition of K, h, and V).

Correction to the notes for 7.1: the note about shared libraries for wrong. Previously, three libraries were built, but each could functi independently. For example, the pcreposix library also included all functions from the basic pcre library. The change is that the three are no longer independent. They are like the Unix libraries. To use pcreposix functions, for example, you need to link with both the pcr the basic pcre library.

Some more features from Perl 5.10 have been added:

(?-n) and (?+n) relative references for recursion and subroutines.

(?(-n) and (?(+n) relative references as conditions.

 $k{name}$ and $g{name}$ are synonyms for k<name>.

\K to reset the start of the matched string; for example, (foo)\Kb
matches bar preceded by foo, but only sets bar as the matched stri

(?| introduces a group where the capturing parentheses in each alt start from the same number; for example, (?|(abc)|(xyz)) sets capt parentheses number 1 in both cases.

h, H, v, V match horizontal and vertical whitespace, respectiv

Release 7.1 24-Apr-07

There is only one new feature in this release: a linebreak setting c PCRE_NEWLINE_ANYCRLF. It is a cut-down version of PCRE_NEWLINE_ANY, recognizes only CRLF, CR, and LF as linebreaks.

A few bugs are fixed (see ChangeLog for details), but the major chan complete re-implementation of the build system. This now has full Au support and so is now "standard" in some sense. It should help with PCRE in a wide variety of environments. NOTE: when building shared libraries for Windows, three dlls are now called libpcre, libpcreposix, and libpcrecpp. Previously, everything included in a single dll.

Another important change is that the dftables auxiliary program is n compiled and run at "make" time by default. Instead, a default set c tables (assuming ASCII coding) is used. If you want to use dftables the character tables as previously, add --enable-rebuild-chartables "configure" command. You must do this if you are compiling PCRE to r system that uses EBCDIC code.

There is a discussion about character tables in the README file. The not to use dftables so that that there is no problem when cross-comp

Release 7.0 19-Dec-06

This release has a new major number because there have been some int upheavals to facilitate the addition of new optimizations and other and to make subsequent maintenance and extension easier. Compilation to be a bit slower, but there should be no major effect on runtime p Previously compiled patterns are NOT upwards compatible with this re you have saved compiled patterns from a previous release, you will h re-compile them. Important changes that are visible to users are:

- 1. The Unicode property tables have been updated to Unicode 5.0.0, w some more scripts.
- 2. The option PCRE_NEWLINE_ANY causes PCRE to recognize any Unicode sequence as a newline.
- 3. The R escape matches a single Unicode newline sequence as a sing
- 4. New features that will appear in Perl 5.10 are now in PCRE. These alternative Perl syntax for named parentheses, and Perl syntax fo recursion.
- 5. The C++ wrapper interface has been extended by the addition of a QuoteMeta function and the ability to allow copy construction and assignment.

For a complete list of changes, see the ChangeLog file.

Release 6.7 04-Jul-06

The main additions to this release are the ability to use the same n multiple sets of parentheses, and support for CRLF line endings in b

library and pcregrep (and in pcretest for testing).

Thanks to Ian Taylor, the stack usage for many kinds of pattern has significantly reduced for certain subject strings.

Release 6.5 01-Feb-06

Important changes in this release:

- 1. A number of new features have been added to pcregrep.
- 2. The Unicode property tables have been updated to Unicode 4.1.0, a supported properties have been extended with script names such as and the derived properties "Any" and "L&". This has necessitated the interal format of compiled patterns. Any saved compiled patterns use \p or \P must be recompiled.
- 3. The specification of recursion in patterns has been changed so th recursive subpatterns are automatically treated as atomic groups. example, (?R) is treated as if it were (?>(?R)). This is necessar otherwise there are situations where recursion does not work.

See the ChangeLog for a complete list of changes, which include a nu fixes and tidies.

Release 6.0 07-Jun-05

The release number has been increased to 6.0 because of the addition major new pieces of functionality.

A new function, pcre_dfa_exec(), which implements pattern matching u algorithm, has been added. This has a number of advantages for certa though it does run more slowly, and lacks the ability to capture sub the other hand, it does find all matches, not just the first, and it better for partial matching. The pcrematching man page discusses the differences.

The pcretest program has been enhanced so that it can make use of th pcre_dfa_exec() matching function and the extra features it provides

The distribution now includes a C++ wrapper library. This is built automatically if a C++ compiler is found. The pcrecpp man page discu interface.

The code itself has been re-organized into many more files, one for function, so it no longer requires everything to be linked in when s

linkage is used. As a consequence, some internal functions have had their names exposed. These functions all have names starting with _p are undocumented, and are not intended for use by outside callers.

The pcregrep program has been enhanced with new functionality such a multiline-matching and options for output more matching context. See ChangeLog for a complete list of changes to the library and the util programs.

Release 5.0 13-Sep-04

The licence under which PCRE is released has been changed to the mor conventional "BSD" licence.

In the code, some bugs have been fixed, and there are also some majc in this release (which is why I've increased the number to 5.0). Som are internal rearrangements, and some provide a number of new facili new features are:

- 1. There's an "automatic callout" feature that inserts callouts before item in the regex, and there's a new callout field that gives the in the pattern useful for debugging and tracing.
- 2. The extra_data structure can now be used to pass in a set of char tables at exec time. This is useful if compiled regex are saved a at a later time when the tables may not be at the same address. I default internal tables are used, the pointer saved with the comp pattern is now set to NULL, which means that you don't need to do special unless you are using custom tables.
- 3. It is possible, with some restrictions on the content of the rege request "partial" matching. A special return code is given if all subject string matched part of the regex. This could be useful fo an input field as it is being typed.
- 4. There is now some optional support for Unicode character properti means that the patterns items such as \p{Lu} and \X can now be us the general category properties are supported. If PCRE is compile support, an additional 90K data structure is include, which incre size of the library dramatically.
- 5. There is support for saving compiled patterns and re-using them 1
- 6. There is support for running regular expressions that were compil different host with the opposite endianness.
- 7. The pcretest program has been extended to accommodate the new fea

The main internal rearrangement is that sequences of literal charact longer handled as strings. Instead, each character is handled on its makes some UTF-8 handling easier, and makes the support of partial m possible. Compiled patterns containing long literal strings will be result of this change; I hope that performance will not be much affe

Release 4.5 01-Dec-03

Again mainly a bug-fix and tidying release, with only a couple of ne

1. It's possible now to compile PCRE so that it does not use recursi function calls when matching. Instead it gets memory from the heap. things down, but may be necessary on systems with limited stacks.

2. UTF-8 string checking has been tightened to reject overlong seque check that a starting offset points to the start of a character. Fai latter returns a new error code: PCRE_ERROR_BADUTF8_OFFSET.

3. PCRE can now be compiled for systems that use EBCDIC code.

Release 4.4 21-Aug-03

This is mainly a bug-fix and tidying release. The only new feature i checks UTF-8 strings for validity by default. There is an option to this, just in case anybody wants that teeny extra bit of performance

Releases 4.1 - 4.3

Sorry, I forgot about updating the NEWS file for these releases. Ple look at ChangeLog.

Release 4.0 17-Feb-03

There have been a lot of changes for the 4.0 release, adding additio functionality and mending bugs. Below is a list of the highlights of functionality. For full details of these features, please consult th documentation. For a complete list of changes, see the ChangeLog fil

1. Support for Perl's $Q... \in B$ escapes.

2. "Possessive quantifiers" ?+, *+, ++, and {,}+ which come from Sun package. They provide some syntactic sugar for simple cases of "atom

grouping".

3. Support for the \G assertion. It is true when the current matchin is at the start point of the match.

4. A new feature that provides some of the functionality that Perl p with (?{...}). The facility is termed a "callout". The way it is don is for the caller to provide an optional function, by setting pcre_c its entry point. To get the function called, the regex must include appropriate points.

5. Support for recursive calls to individual subpatterns. This makes easy to get totally confused.

6. Support for named subpatterns. The Python syntax (?P<name>...) is name a group.

7. Several extensions to UTF-8 support; it is now fairly complete. T option for pcregrep to make it operate in UTF-8 mode.

8. The single man page has been split into a number of separate man These also give rise to individual HTML pages which are put in a sep directory. There is an index.html page that lists them all. Some hyp between the pages has been installed.

Release 3.5 15-Aug-01

1. The configuring system has been upgraded to use later versions of and libtool. By default it builds both a shared and a static library supports it. You can use --disable-shared or --disable-static on the command if you want only one of them.

2. The pcretest utility is now installed along with pcregrep because useful for users (to test regexs) and by doing this, it automaticall relinked by libtool. The documentation has been turned into a man pa there are now .1, .txt, and .html versions in /doc.

3. Upgrades to pcregrep:

(i) Added long-form option names like gnu grep.

(ii) Added --help to list all options with an explanatory phrase

(iii) Added -r, --recursive to recurse into sub-directories.

(iv) Added -f, --file to read patterns from a file.

4. Added --enable-newline-is-cr and --enable-newline-is-lf to the cc script, to force use of CR or LF instead of \n in the source. On non systems, the value can be set in config.h.

5. The limit of 200 on non-capturing parentheses is a _nesting_ limi

absolute limit. Changed the text of the error message to make this c likewise updated the man page.

6. The limit of 99 on the number of capturing subpatterns has been r The new limit is 65535, which I hope will not be a "real" limit.

Release 3.3 01-Aug-00

There is some support for UTF-8 character strings. This is incomplet experimental. The documentation describes what is and what is not im Otherwise, this is just a bug-fixing release.

Release 3.0 01-Feb-00

1. A "configure" script is now used to configure PCRE for Unix syste builds a Makefile, a config.h file, and the pcre-config script.

2. PCRE is built as a shared library by default.

3. There is support for POSIX classes such as [:alpha:].

5. There is an experimental recursion feature.

IMPORTANT FOR THOSE UPGRADING FROM VERSIONS BEFORE 2.00

Please note that there has been a change in the API such that a larg ovector is required at matching time, to provide some additional wor The new man page has details. This change was necessary in order to some of the new functionality in Perl 5.005.

IMPORTANT FOR THOSE UPGRADING FROM VERSION 2.00

Another (I hope this is the last!) change has been made to the API f pcre_compile() function. An additional argument has been added to ma possible to pass over a pointer to character tables built in the cur locale by pcre_maketables(). To use the default tables, this new arg should be passed as NULL.

IMPORTANT FOR THOSE UPGRADING FROM VERSION 2.05

Yet another (and again I hope this really is the last) change has be to the API for the pcre_exec() function. An additional argument has added to make it possible to start the match other than at the start subject string. This is important if there are lookbehinds. The new page has the details, but you just want to convert existing programs you need to do is to stick in a new fifth argument to pcre_exec(), w value of zero. For example, change

pcre_exec(pattern, extra, subject, length, options, ovec, ovecsize
to
 pcre_exec(pattern, extra, subject, length, 0, options, ovec, ovecs

* * * *

ChangeLog for PCRE

Version 7.2 19-June-07

- 1. If the fr_FR locale cannot be found for test 3, try the "french" which is apparently normally available under Windows.
- 2. Re-jig the pcregrep tests with different newline settings in an to make them independent of the local environment's newline sett
- 3. Add code to configure.ac to remove -g from the CFLAGS default se
- 4. Some of the "internals" tests were previously cut out when the l was not 2, because the output contained actual offsets. The rece "Z" feature of pcretest means that these can be cut out, making usable with all link sizes.
- 5. Implemented Stan Switzer's goto replacement for longjmp() when n stack recursion. This gives a massive performance boost under BS a small improvement under Linux. However, it saves one field in in all cases.
- 6. Added more features from the forthcoming Perl 5.10:
 - (a) (?-n) (where n is a string of digits) is a relative subrouti recursion call. It refers to the nth most recently opened pa
 - (b) (?+n) is also a relative subroutine call; it refers to the n to be opened parentheses.
 - (c) Conditions that refer to capturing parentheses can be specif relatively, for example, (?(-2)... or (?(+3)...
 - (d) \K resets the start of the current match so that everything is not part of it.
 - (e) \k{name} is synonymous with \k<name> and \k'name' (.NET comp
 - (f) \g{name} is another synonym part of Perl 5.10's unificatio reference syntax.
 - (g) (?| introduces a group in which the numbering of parentheses alternative starts with the same number.
 - (h) h, H, v, and V match horizontal and vertical whitespace.
- 7. Added two new calls to pcre_fullinfo(): PCRE_INFO_OKPARTIAL and

PCRE_INFO_JCHANGED.

- 8. A pattern such as (.*(.)?)* caused pcre_exec() to fail by eith terminating or by crashing. Diagnosed by Viktor Griph; it was in for detecting groups that can match an empty string.
- 9. A pattern with a very large number of alternatives (more than se hundred) was running out of internal workspace during the pre-cc phase, where pcre_compile() figures out how much memory will be bit of new cunning has reduced the workspace needed for groups w alternatives. The 1000-alternative test pattern now uses 12 byte workspace instead of running out of the 4096 that are available.
- 10. Inserted some missing (unsigned int) casts to get rid of compile
- 11. Applied patch from Google to remove an optimization that didn't The report of the bug said:

pcrecpp::RE("a*").FullMatch("aaa") matches, while pcrecpp::RE("a*?").FullMatch("aaa") does not, and pcrecpp::RE("a*?\\z").FullMatch("aaa") does again.

12. If \p or \P was used in non-UTF-8 mode on a character greater th it matched the wrong number of bytes.

Version 7.1 24-Apr-07

- 1 Applied Bob Rossi and Daniel G's pat
- 1. Applied Bob Rossi and Daniel G's patches to convert the build sy that is more "standard", making use of automake and other Autotc is some re-arrangement of the files and adjustment of comments c on this.
- 2. Part of the patch fixed a problem with the pcregrep tests. The t for recursive directory scanning broke on some systems because t are not scanned in any specific order and on different systems t was different. A call to "sort" has been inserted into RunGrepTe approprate test as a short-term fix. In the longer term there ma alternative.
- 3. I had an email from Eric Raymond about problems translating some man pages to HTML (despite the fact that I distribute HTML pages people do their own conversions for various reasons). The proble concerned the use of low-level troff macros .br and .in. I have removed all such uses from the man pages (some were redundant, s be replaced by .nf/.fi pairs). The 132html script that I use to HTML has been updated to handle .nf/.fi and to complain if it en .br or .in.

- 4. Updated comments in configure.ac that get placed in config.h.in arranged for config.h to be included in the distribution, with t config.h.generic, for the benefit of those who have to compile w Autotools (compare pcre.h, which is now distributed as pcre.h.ge
- 5. Updated the support (such as it is) for Virtual Pascal, thanks t Weber: (1) pcre_internal.h was missing some function renames; (2 makevp.bat for the current PCRE, using the additional files makevp_c.txt, makevp_l.txt, and pcregexp.pas.
- 6. A Windows user reported a minor discrepancy with test 2, which t to be caused by a trailing space on an input line that had got l copy. The trailing space was an accident, so I've just removed i
- 7. Add -Wl,-R... flags in pcre-config.in for *BSD* systems, as I'm that is needed.
- 8. Mark ucp_table (in ucptable.h) and ucp_gentype (in pcre_ucp_sear as "const" (a) because they are and (b) because it helps the PHP maintainers who have recently made a script to detect big data s in the php code that should be moved to the .rodata section. I r to update Builducptable as well, so it won't revert if ucptable. re-created.
- 9. Added some extra #ifdef SUPPORT_UTF8 conditionals into pcretest. pcre_printint.src, pcre_compile.c, pcre_study.c, and pcre_tables order to be able to cut out the UTF-8 tables in the latter when support is not required. This saves 1.5-2K of code, which is imp some applications.

Later: more #ifdefs are needed in pcre_ord2utf8.c and pcre_valid so as not to refer to the tables, even though these functions wi called when UTF-8 support is disabled. Otherwise there are probl shared library.

- 10. Fixed two bugs in the emulated memmove() function in pcre_intern
 - (a) It was defining its arguments as char * instead of void *.
 - (b) It was assuming that all moves were upwards in memory; this a long time ago when I wrote it, but is no longer the case.

The emulated memove() is provided for those environments that ha memmove() nor bcopy(). I didn't think anyone used it these days, is clearly not the case, as these two bugs were recently reporte

11. The script PrepareRelease is now distributed: it calls 132html, and Detrail to create the HTML documentation, the .txt form of t pages, and it removes trailing spaces from listed files. It also pcre.h.generic and config.h.generic from pcre.h and config.h. In case, it wraps all the #defines with #ifndefs. This script shoul before "make dist".

- 12. Fixed two fairly obscure bugs concerned with quantified caseless with Unicode property support.
 - (a) For a maximizing quantifier, if the two different cases of t character were of different lengths in their UTF-8 codings (some cases like this - I found 11), and the matching functio back up over a mixture of the two cases, it incorrectly assu were both the same length.
 - (b) When PCRE was configured to use the heap rather than the sta recursion during matching, it was not correctly preserving t the other case of a UTF-8 character when checking ahead for while processing a minimizing repeat. If the check also invo matching a wide character, but failed, corruption could caus erroneous result when trying to check for a repeat of the or character.
- 13. Some tidying changes to the testing mechanism:
 - (a) The RunTest script now detects the internal link size and wh is UTF-8 and UCP support by running ./pcretest -C instead of values substituted by "configure". (The RunGrepTest script a this for UTF-8.) The configure.ac script no longer substitut relevant variables.
 - (b) The debugging options /B and /D in pcretest show the compile with length and offset values. This means that the output is for different internal link sizes. Test 2 is skipped for lin other than 2 because of this, bypassing the problem. Unfortu there was also a test in test 3 (the locale tests) that used failed for link sizes other than 2. Rather than cut the whol I have added a new /Z option to pcretest that replaces the l offset values with spaces. This is now used to make test 3 i of link size. (Test 2 will be tidied up later.)
- 14. If erroroffset was passed as NULL to pcre_compile, it provoked a segmentation fault instead of returning the appropriate error me
- 15. In multiline mode when the newline sequence was set to "any", th ^\$ would give a match between the \r and \n of a subject such as This doesn't seem right; it now treats the CRLF combination as t ending, and so does not match in that case. It's only a pattern that would hit this one: something like ^ABC\$ would have failed and then tried again after \r\n.
- 16. Changed the comparison command for RunGrepTest from "diff -u" to in an attempt to make files that differ only in their line termi

compare equal. This works on Linux.

- 17. Under certain error circumstances pcregrep might try to free ran as it exited. This is now fixed, thanks to valgrind.
- 19. In pcretest, if the pattern /(?m)^\$/g<any> was matched against t "abc\r\n\r\n", it found an unwanted second match after the secon was because its rules for how to advance for /g after matching a string at the end of a line did not allow for this case. They no it specially.
- 20. pcretest is supposed to handle patterns and data of any length, extending its buffers when necessary. It was getting this wrong buffer for a data line had to be extended.
- 21. Added PCRE_NEWLINE_ANYCRLF which is like ANY, but matches only C CRLF as a newline sequence.
- 22. Code for handling Unicode properties in pcre_dfa_exec() wasn't b out by #ifdef SUPPORT_UCP. This did no harm, as it could never b I have nevertheless tidied it up.
- 23. Added some casts to kill warnings from HP-UX ia64 compiler.
- 24. Added a man page for pcre-config.

Version 7.0 19-Dec-06

- 1. Fixed a signed/unsigned compiler warning in pcre_compile.c, show moving to gcc 4.1.1.
- 2. The -S option for pcretest uses setrlimit(); I had omitted to #i sys/time.h, which is documented as needed for this function. It seem to matter on Linux, but it showed up on some releases of OS
- 3. It seems that there are systems where bytes whose values are gre 127 match isprint() in the "C" locale. The "C" locale should be default when a C program starts up. In most systems, only ASCII characters match isprint(). This difference caused the output fr to vary, making some of the tests fail. I have changed pcretest
 - (a) When it is outputting text in the compiled version of a patt other than 32-126 are always shown as hex escapes.
 - (b) When it is outputting text that is a matched part of a subje it does the same, unless a different locale has been set for (using the /L modifier). In this case, it uses isprint() to

4. Fixed a major bug that caused incorrect computation of the amoun required for a compiled pattern when options that changed within pattern affected the logic of the preliminary scan that determin length. The relevant options are -x, and -i in UTF-8 mode. The r that the computed length was too small. The symptoms of this bug either the PCRE error "internal error: code overflow" from pcre_ or a glibc crash with a message such as "pcretest: free(): inval size (fast)". Examples of patterns that provoked this bug (shown pcretest format) are:

/(?-x:)/x /(?x)(?-x: \s*#\s*)/ /((?i)[\x{c0}])/8 /(?i:[\x{c0}])/8

HOWEVER: Change 17 below makes this fix obsolete as the memory c is now done differently.

- 5. Applied patches from Google to: (a) add a QuoteMeta function to wrapper classes; (b) implement a new function in the C++ scanner more efficient than the old way of doing things because it avoid recursion in the regex matching; (c) add a paragraph to the docu for the FullMatch() function.
- 6. The escape sequence \n was being treated as whatever was defined "newline". Not only was this contrary to the documentation, whic that \n is character 10 (hex 0A), but it also went horribly wron "newline" was defined as CRLF. This has been fixed.
- 7. In pcre_dfa_exec.c the value of an unsigned integer (the variabl was being set to -1 for the "end of line" case (supposedly a val character can have). Though this value is never used (the check line is "zero bytes in current character"), it caused compiler c I've changed it to 0xffffffff.
- 8. In pcre_version.c, the version string was being built by a seque C macros that, in the event of PCRE_PRERELEASE being defined as string (as it is for production releases) called a macro with an argument. The C standard says the result of this is undefined. T compiler treats it as an empty string (which was what was wanted reported that Visual C gives an error. The source has been hacke avoid this problem.
- 9. On the advice of a Windows user, included <io.h> and <fcntl.h> i builds of pcretest, and changed the call to _setmode() to use _C instead of 0x8000. Made all the #ifdefs test both _WIN32 and WIN of them did).
- 10. Originally, pcretest opened its input and output without "b"; th told that "b" was needed in some environments, so it was added f

5.0 to both the input and output. (It makes no difference on Uni systems.) Later I was told that it is wrong for the input on Win now abstracted the modes into two macros, to make it easier to f them, and removed "b" from the input mode under Windows.

- 11. Added pkgconfig support for the C++ wrapper library, libpcrecpp.
- 12. Added -help and --help to pcretest as an official way of being r of the options.
- 13. Removed some redundant semicolons after macro calls in pcrecppar and pcrecpp.cc because they annoy compilers at high warning leve
- 14. A bit of tidying/refactoring in pcre_exec.c in the main bumpalon
- 15. Fixed an occurrence of == in configure.ac that should have been scripts are not C programs :-) and which was not noticed because on Linux.
- 16. pcretest is supposed to handle any length of pattern and data li line or as a continued sequence of lines) by extending its input necessary. This feature was broken for very long pattern lines, a string of junk being passed to pcre_compile() if the pattern w than about 50K.
- 17. I have done a major re-factoring of the way pcre_compile() compu amount of memory needed for a compiled pattern. Previously, ther that made a preliminary scan of the pattern in order to do this. OK when PCRE was new, but as the facilities have expanded, it ha harder and harder to keep it in step with the real compile phase have been a number of bugs (see for example, 4 above). I have no cunning way of running the real compile function in a "fake" mod enables it to compute how much memory it would need, while actua ever using a few hundred bytes of working memory and without too tests of the mode. This should make future maintenance and devel easier. A side effect of this work is that the limit of 200 on t depth of parentheses has been removed (though this was never a s limitation, I suspect). However, there is a downside: pcre_compi runs more slowly than before (30% or more, depending on the patt hope this isn't a big issue. There is no effect on runtime perfo
- 18. Fixed a minor bug in pcretest: if a pattern line was not termina newline (only possible for the last line of a file) and it was a pattern that set a locale (followed by /Lsomething), pcretest cr
- 19. Added additional timing features to pcretest. (1) The -tm option matching only, not compiling. (2) Both -t and -tm can be followe separate command line item, by a number that specifies the numbe repeats to use when timing. The default is 50000; this gives bet precision, but takes uncomfortably long for very large patterns.

- 20. Extended pcre_study() to be more clever in cases where a branch subpattern has no definite first character. For example, (a*|b*) previously give no result from pcre_study(). Now it recognizes t first character must be a, b, c, or d.
- 21. There was an incorrect error "recursive call could loop indefini a subpattern (or the entire pattern) that was being tested for m empty string contained only one non-empty item after a nested su For example, the pattern (?>\x{100}*)\d(?R) provoked this error incorrectly, because the \d was being skipped in the check.
- 22. The pcretest program now has a new pattern option /B and a comma option -b, which is equivalent to adding /B to every pattern. Th it to show the compiled bytecode, without the additional informa -d shows. The effect of -d is now the same as -b with -i (and si is the same as /B/I).
- 23. A new optimization is now able automatically to treat some seque as a*b as a*+b. More specifically, if something simple (such as or a simple class like \d) has an unlimited quantifier, and is f something that cannot possibly match the quantified thing, the q is automatically "possessified".
- 24. A recursive reference to a subpattern whose number was greater t went wrong under certain circumstances in UTF-8 mode. This bug c have affected the operation of pcre_study().
- 25. Realized that a little bit of performance could be had by replac (c & 0xc0) == 0xc0 with c >= 0xc0 when processing UTF-8 characte
- 26. Timing data from pcretest is now shown to 4 decimal places inste
- 27. Possessive quantifiers such as a++ were previously implemented b them into atomic groups such as (\$>a+). Now they have their own which improves performance. This includes the automatically crea from 23 above.
- 28. A pattern such as (?=(\w+))\1: which simulates an atomic group u lookahead was broken if it was not anchored. PCRE was mistakenly the first matched character to be a colon. This applied both to numbered groups.
- 29. The ucpinternal.h header file was missing its idempotency #ifdef
- 30. I was sent a "project" file called libpcre.a.dev which I underst building PCRE on Windows easier, so I have included it in the di
- 31. There is now a check in pcretest against a ridiculously large nu returned by pcre_exec() or pcre_dfa_exec(). If this happens in a

loop, the loop is abandoned.

- 32. Forward references to subpatterns in conditions such as (?(2)... subpattern 2 is defined later cause pcre_compile() to search for the pattern for the relevant set of parentheses. This search wen when there were unescaped parentheses in a character class, pare escaped with \Q...\E, or parentheses in a #-comment in /x mode.
- 33. "Subroutine" calls and backreferences were previously restricted referencing subpatterns earlier in the regex. This restriction h been removed.
- 34. Added a number of extra features that are going to be in Perl 5. whole, these are just syntactic alternatives for features that P previously implemented using the Python syntax or my own inventi other formats are all retained for compatibility.
 - (a) Named groups can now be defined as (?<name>...) or (?'name'. as (?P<name>...). The new forms, as well as being in Perl 5. also .NET compatible.
 - (b) A recursion or subroutine call to a named group can now be d (?&name;) as well as (?P>name).
 - (c) A backreference to a named group can now be defined as \k<na \k'name' as well as (?P=name). The new forms, as well as bei 5.10, are also .NET compatible.
 - (d) A conditional reference to a named group can now use the syn (?(<name>) or (?('name') as well as (?(name).
 - (e) A "conditional group" of the form (?(DEFINE)...) can be used groups (named and numbered) that are never evaluated inline, called as "subroutines" from elsewhere. In effect, the DEFIN is always false. There may be only one alternative in such a
 - (f) A test for recursion can be given as (?(R1).. or (?(R&name;) as the simple (?(R). The condition is true only if the most recursion is that of the given number or name. It does not s through the entire recursion stack.
 - (g) The escape gN or $g\{N\}$ has been added, where N is a positiv negative number, specifying an absolute or relative referenc
- 35. Tidied to get rid of some further signed/unsigned compiler warni some "unreachable code" warnings.
- 36. Updated the Unicode property tables to Unicode version 5.0.0. Am things, this adds five new scripts.

- 37. Perl ignores orphaned \E escapes completely. PCRE now does the s There were also incompatibilities regarding the handling of \Q.. character classes, for example with patterns like [\Qa\E-\Qz\E] hyphen was adjacent to \Q or \E. I hope I've cleared all this up
- 38. Like Perl, PCRE detects when an indefinitely repeated parenthesi matches an empty string, and forcibly breaks the loop. There wer this code in non-simple cases. For a pattern such as ^(a()*)* against aaaa the result was just "a" rather than "aaaa", for e separate and independent bugs (that affected different cases) ha fixed.
- 39. Refactored the code to abolish the use of different opcodes for capturing bracket numbers. This is a tidy that I avoided doing w removed the limit on the number of capturing brackets for 3.5 ba The new approach is not only tidier, it makes it possible to red memory needed to fix the previous bug (38).
- 40. Implemented PCRE_NEWLINE_ANY to recognize any of the Unicode new sequences (http://unicode.org/unicode/reports/tr18/) as "newline processing dot, circumflex, or dollar metacharacters, or #-comme mode.
- 41. Add \R to match any Unicode newline sequence, as suggested in th report.
- 42. Applied patch, originally from Ari Pollak, modified by Google, t copy construction and assignment in the C++ wrapper.
- 43. Updated pcregrep to support "--newline=any". In the process, I f couple of bugs that could have given wrong results in the "--new case.
- 44. Added a number of casts and did some reorganization of signed/un variables following suggestions from Dair Grant. Also renamed th "this" as "item" because it is a C++ keyword.
- 45. Arranged for dftables to add

#include "pcre_internal.h"

to pcre_chartables.c because without it, gcc 4.x may remove the definition from the final binary if PCRE is built into a static dead code stripping is activated.

46. For an unanchored pattern, if a match attempt fails at the start newline sequence, and the newline setting is CRLF or ANY, and th characters are CRLF, advance by two characters instead of one. Version 6.7 04-Jul-06

- 1. In order to handle tests when input lines are enormously long, p been re-factored so that it automatically extends its buffers wh necessary. The code is crude, but this _is_ just a test program. default size has been increased from 32K to 50K.
- The code in pcre_study() was using the value of the re argument testing it for NULL. (Of course, in any sensible call of the fun won't be NULL.)
- 3. The memmove() emulation function in pcre_internal.h, which is us systems that lack both memmove() and bcopy() - that is, hardly e was missing a "static" storage class specifier.
- 4. When UTF-8 mode was not set, PCRE looped when compiling certain containing an extended class (one that cannot be represented by because it contains high-valued characters or Unicode property i [\pZ]). Almost always one would set UTF-8 mode when processing s pattern, but PCRE should not loop if you do not (it no longer dc [Detail: two cases were found: (a) a repeated subpattern contain extended class; (b) a recursive reference to a subpattern that f previous extended class. It wasn't skipping over the extended cl correctly when UTF-8 mode was not set.]
- 5. A negated single-character class was not being recognized as fix in lookbehind assertions such as (?<=[^f]), leading to an incorr compile error "lookbehind assertion is not fixed length".
- 6. The RunPerlTest auxiliary script was showing an unexpected diffe between PCRE and Perl for UTF-8 tests. It turns out that it is h write a Perl script that can interpret lines of an input file ei byte characters or as UTF-8, which is what "perltest" was being do for the non-UTF-8 and UTF-8 tests, respectively. Essentially can't do is switch easily at run time between having the "use ut or not. In the end, I fudged it by using the RunPerlTest script "use utf8;" explicitly for the UTF-8 tests.
- 7. In multiline (/m) mode, PCRE was matching ^ after a terminating the end of the subject string, contrary to the documentation and Perl does. This was true of both matching functions. Now it matc the start of the subject and immediately after *internal* newlin
- 8. A call of pcre_fullinfo() from pcretest to get the option bits w a pointer to an int instead of a pointer to an unsigned long int caused problems on 64-bit systems.
- 9. Applied a patch from the folks at Google to pcrecpp.cc, to fix " instance of the 'standard' template library not being so standar

- 10. There was no check on the number of named subpatterns nor the ma length of a subpattern name. The product of these values is used the size of the memory block for a compiled pattern. By supplyin long subpattern name and a large number of named subpatterns, th computation could be caused to overflow. This is now prevented b the length of names to 32 characters, and the number of named su to 10,000.
- 11. Subpatterns that are repeated with specific counts have to be re the compiled pattern. The size of memory for this was computed f length of the subpattern and the repeat count. The latter is lim 65535, but there was no limit on the former, meaning that intege could in principle occur. The compiled length of a repeated subp now limited to 30,000 bytes in order to prevent this.
- 12. Added the optional facility to have named substrings with the sa
- 13. Added the ability to use a named substring as a condition, using Python syntax: (?(name)yes|no). This overloads (?(R)... and name are numbers (not recommended). Forward references are permitted.
- 14. Added forward references in named backreferences (if you see wha
- 15. In UTF-8 mode, with the PCRE_DOTALL option set, a quantified dot pattern could run off the end of the subject. For example, the p "(?s)(.{1,5})"8 did this with the subject "ab".
- 16. If PCRE_DOTALL or PCRE_MULTILINE were set, pcre_dfa_exec() behav PCRE_CASELESS was set when matching characters that were quantif or *.
- 17. A character class other than a single negated character that had but no maximum quantifier - for example [ab]{6,} - was not handl correctly by pce_dfa_exec(). It would match only one character.
- 18. A valid (though odd) pattern that looked like a POSIX character class but used an invalid character after [(for example [[,abc, pcre_compile() to give the error "Failed: internal error: code c in some cases to crash with a glibc free() error. This could eve the pattern terminated after [[but there just happened to be a letters, a binary zero, and a closing] in the memory that follo
- 19. Perl's treatment of octal escapes in the range \400 to \777 has over the years. Originally (before any Unicode support), just th bits were taken. Thus, for example, \500 really meant \100. Nowa output from "man perlunicode" includes this:

The regular expression compiler produces polymorphic opcodes. is, the pattern adapts to the data and automatically switches

the Unicode character scheme when presented with Unicode datainstead uses a traditional byte scheme when presented with byt data.

Sadly, a wide octal escape does not cause a switch, and in a str no other multibyte characters, these octal escapes are treated a Thus, in Perl, the pattern /\500/ actually matches \100 but the /\500|\x{1ff}/ matches \500 or 777 because the whole thing is t Unicode string.

I have not perpetrated such confusion in PCRE. Up till now, it t the bottom 8 bits, as in old Perl. I have now made octal escapes values greater than \377 illegal in non-UTF-8 mode. In UTF-8 mod translate to the appropriate multibyte character.

- 29. Applied some refactoring to reduce the number of warnings from M and Borland compilers. This has included removing the fudge intr seven years ago for the OS/2 compiler (see 2.02/2 below) because a warning about an unused variable.
- 21. PCRE has not included VT (character 0x0b) in the set of whitespa characters since release 4.0, because Perl (from release 5.004) [Or at least, is documented not to: some releases seem to be in with the documentation.] However, when a pattern was studied wit pcre_study() and all its branches started with \s, PCRE still in as a possible starting character. Of course, this did no harm; i caused an unnecessary match attempt.
- 22. Removed a now-redundant internal flag bit that recorded the fact dependency changed within the pattern. This was once needed for byte" processing, but is no longer used. This recovers a now-sca bit. Also moved the least significant internal flag bit to the m significant bit of the word, which was not previously used (hang the days when it was an int rather than a uint) to free up anoth the future.
- 23. Added support for CRLF line endings as well as CR and LF. As wel default being selectable at build time, it can now be changed at via the PCRE_NEWLINE_xxx flags. There are now options for pcregr specify that it is scanning data with non-default line endings.
- 24. Changed the definition of CXXLINK to make it agree with the defi LINK in the Makefile, by replacing LDFLAGS to CXXFLAGS.
- 25. Applied Ian Taylor's patches to avoid using another stack frame recursions. This makes a big different to stack usage for some p
- 26. If a subpattern containing a named recursion or subroutine refer as (?P>B) was quantified, for example (xxx(?P>B)){3}, the calcul the space required for the compiled pattern went wrong and gave

value. Depending on the environment, this could lead to "Failed: error: code overflow at offset 49" or "glibc detected double fre corruption" errors.

- 27. Applied patches from Google (a) to support the new newline modes advance over multibyte UTF-8 characters in GlobalReplace.
- 28. Change free() to pcre_free() in pcredemo.c. Apparently this make difference for some implementation of PCRE in some Windows versi
- 29. Added some extra testing facilities to pcretest:

\q<number> in a data line sets the "match limit" value \Q<number> in a data line sets the "match recursion limt" valu -S <number> sets the stack size, where <number> is in megabytes

The -S option isn't available for Windows.

Version 6.6 06-Feb-06

- 1. Change 16(a) for 6.5 broke things, because PCRE_DATA_SCOPE was n in pcreposix.h. I have copied the definition from pcre.h.
- 2. Change 25 for 6.5 broke compilation in a build directory out-ofbecause pcre.h is no longer a built file.
- 3. Added Jeff Friedl's additional debugging patches to pcregrep. Th not normally included in the compiled code.

Version 6.5 01-Feb-06

- When using the partial match feature with pcre_dfa_exec(), it wa anchoring the second and subsequent partial matches at the new s point. This could lead to incorrect results. For example, with t /1234/, partially matching against "123" and then "a4" gave a ma
- 2. Changes to pcregrep:
 - (a) All non-match returns from pcre_exec() were being treated as to match the line. Now, unless the error is PCRE_ERROR_NOMAT error message is output. Some extra information is given for PCRE_ERROR_MATCHLIMIT and PCRE_ERROR_RECURSIONLIMIT errors, probably the only errors that are likely to be caused by use specifying a regex that has nested indefinite repeats, for i If there are more than 20 of these errors, pcregrep is aband

- (b) A binary zero was treated as data while matching, but termin output line if it was written out. This has been fixed: bina are now no different to any other data bytes.
- (c) Whichever of the LC_ALL or LC_CTYPE environment variables is used to set a locale for matching. The --locale=xxxx long op been added (no short equivalent) to specify a locale explici pcregrep command, overriding the environment variables.
- (d) When -B was used with -n, some line numbers in the output we than they should have been.
- (e) Added the -o (--only-matching) option.
- (f) If -A or -C was used with -c (count only), some lines of con accidentally printed for the final match.
- (g) Added the -H (--with-filename) option.
- (h) The combination of options -rh failed to suppress file names that were found from directory arguments.
- (i) Added the -D (--devices) and -d (--directories) options.
- (j) Added the -F (--fixed-strings) option.
- (k) Allow "-" to be used as a file name for -f as well as for a
- (1) Added the --colo(u)r option.
- (m) Added Jeffrey Friedl's -S testing option, but within #ifdefs is not present by default.
- 3. A nasty bug was discovered in the handling of recursive patterns items such as (?R) or (?1), when the recursion could match a num alternatives. If it matched one of the alternatives, but subsequ outside the recursion, there was a failure, the code tried to ba the recursion. However, because of the way PCRE is implemented, possible, and the result was an incorrect result from the match.

In order to prevent this happening, the specification of recursi been changed so that all such subpatterns are automatically trea atomic groups. Thus, for example, (?R) is treated as if it were

4. I had overlooked the fact that, in some locales, there are chara which isalpha() is true but neither isupper() nor islower() are the fr_FR locale, for instance, the \xAA and \xBA characters (or and ordfeminine) are like this. This affected the treatment of \ when they appeared in character classes, but not when they appea a character class. The bit map for "word" characters is now crea separately from the results of isalnum() instead of just taking upper, lower, and digit maps. (Plus the underscore character, of

- 5. The above bug also affected the handling of POSIX character clas [[:alpha:]] and [[:alnum:]]. These do not have their own bit map permanent tables. Instead, the bit maps for such a class were pr created as the appropriate unions of the upper, lower, and digit Now they are created by subtraction from the [[:word:]] class, w its own bitmap.
- 6. The [[:blank:]] character class matches horizontal, but not vert It is created by subtracting the vertical space characters (\x09 \x0b, \x0c) from the [[:space:]] bitmap. Previously, however, th subtraction was done in the overall bitmap for a character class that a class such as [\x0c[:blank:]] was incorrect because \x0c be recognized. This bug has been fixed.
- 7. Patches from the folks at Google:

(a) pcrecpp.cc: "to handle a corner case that may or may not h real life, but is still worth protecting against".

(b) pcrecpp.cc: "corrects a bug when negative radixes are used regular expressions".

(c) pcre_scanner.cc: avoid use of std::count() because not all have it.

(d) Split off pcrecpparg.h from pcrecpp.h and had the former b "configure" and the latter not, in order to fix a problem some with compiling the Arg class on HP-UX.

- (e) Improve the error-handling of the C++ wrapper a little bit
- (f) New tests for checking recursion limiting.
- 8. The pcre_memmove() function, which is used only if the environme have a standard memmove() function (and is therefore rarely comp contained two bugs: (a) use of int instead of size_t, and (b) it returning a result (though PCRE never actually uses the result).
- 9. In the POSIX regexec() interface, if nmatch is specified as a ri large number - greater than INT_MAX/(3*sizeof(int)) - REG_ESPACE returned instead of calling malloc() with an overflowing number most likely cause subsequent chaos.
- 10. The debugging option of pcretest was not showing the NO_AUTO_CAP
- 11. The POSIX flag REG_NOSUB is now supported. When a pattern that w with this option is matched, the nmatch and pmatch options of re

ignored.

- 12. Added REG_UTF8 to the POSIX interface. This is not defined by PO provided in case anyone wants to the the POSIX interface with UT strings.
- 13. Added CXXLDFLAGS to the Makefile parameters to provide settings C++ linking (needed for some HP-UX environments).
- 14. Avoid compiler warnings in get_ucpname() when compiled without U (unused parameter) and in the pcre_printint() function (omitted switch label when the default is to do nothing).
- 15. Added some code to make it possible, when PCRE is compiled as a library, to replace subject pointers for pcre_exec() with a smar class, thus making it possible to process discontinuous strings.
- 16. The two macros PCRE_EXPORT and PCRE_DATA_SCOPE are confusing, an much the same function. They were added by different people who to make PCRE easy to compile on non-Unix systems. It has been su that PCRE_EXPORT be abolished now that there is more automatic a for compiling on Windows systems. I have therefore replaced it w PCRE_DATA_SCOPE. This is set automatically for Windows; if not s defaults to "extern" for C or "extern C" for C++, which works fi Unix-like systems. It is now possible to override the value of P SCOPE with something explicit in config.h. In addition:
 - (a) pcreposix.h still had just "extern" instead of either of the I have replaced it with PCRE_DATA_SCOPE.
 - (b) Functions such as _pcre_xclass(), which are internal to the but external in the C sense, all had PCRE_EXPORT in their de This is apparently wrong for the Windows case, so I have rem (It makes no difference on Unix-like systems.)
- 17. Added a new limit, MATCH_LIMIT_RECURSION, which limits the depth of recursive calls to match(). This is different to MATCH_LIMIT that limits the total number of calls to match(), not all of whi the depth of recursion. Limiting the recursion depth limits the stack (or heap if NO_RECURSE is set) that is used. The default c when PCRE is compiled, and changed at run time. A patch from Goc this functionality to the C++ interface.
- 18. Changes to the handling of Unicode character properties:
 - (a) Updated the table to Unicode 4.1.0.
 - (b) Recognize characters that are not in the table as "Cn" (unde
 - (c) I revised the way the table is implemented to a much improve

which includes recognition of ranges. It now supports the ra are defined in UnicodeData.txt, and it also amalgamates othe characters into ranges. This has reduced the number of entri table from around 16,000 to around 3,000, thus reducing its considerably. I realized I did not need to use a tree struct all - a binary chop search is just as efficient. Having redu number of entries, I extended their size from 6 bytes to 8 b allow for more data.

- (d) Added support for Unicode script names via properties such a
- 19. In UTF-8 mode, a backslash followed by a non-Ascii character was matching that character.
- 20. When matching a repeated Unicode property with a minimum greater (for example \pL{2,}), PCRE could look past the end of the subje reached it while seeking the minimum number of characters. This happen only if some of the characters were more than one byte lo there is a check for at least the minimum number of bytes.
- 21. Refactored the implementation of \p and \P so as to be more gene allow for more different types of property in future. This has c compiled form incompatibly. Anybody with saved compiled patterns \p or \P will have to recompile them.
- 22. Added "Any" and "L&" to the supported property types.
- 23. Recognize $x{\ldots}$ as a code point specifier, even when not in UT but give a compile time error if the value is greater than 0xff.
- 24. The man pages for pcrepartial, pcreprecompile, and pcre_compile2 accidentally not being installed or uninstalled.
- 25. The pcre.h file was built from pcre.h.in, but the only changes t made were to insert the current release number. This seemed sill it made things harder for people building PCRE on systems that d "configure". I have turned pcre.h into a distributed file, no lo by "configure", with the version identification directly include no longer a pcre.h.in file.

However, this change necessitated a change to the pcre-config sc well. It is built from pcre-config.in, and one of the substituti release number. I have updated configure.ac so that ./configure the release number by grepping pcre.h.

26. Added the ability to run the tests under valgrind.

Version 6.4 05-Sep-05

- Change 6.0/10/(1) to pcregrep introduced a bug that caused separ "--" to be printed when multiple files were scanned, even when n -A, -B, or -C options were used. This is not compatible with Gnu consider it to be a bug, and have restored the previous behaviou
- 2. A couple of code tidies to get rid of compiler warnings.
- 3. The pcretest program used to cheat by referring to symbols in th whose names begin with _pcre_. These are internal symbols that a really supposed to be visible externally, and in some environmen possible to suppress them. The cheating is now confined to inclu certain files from the library's source, which is a bit cleaner.
- 4. Renamed pcre.in as pcre.h.in to go with pcrecpp.h.in; it also ma file's purpose clearer.
- 5. Reorganized pcre_ucp_findchar().

Version 6.3 15-Aug-05

- 1. The file libpcre.pc.in did not have general read permission in t
- 2. There were some problems when building without C++ support:
 - (a) If C++ support was not built, "make install" and "make test" tried to test it.
 - (b) There were problems when the value of CXX was explicitly set changes have been made to try to fix these, and ...
 - (c) --disable-cpp can now be used to explicitly disable C++ supp
 - (d) The use of @CPP_OBJ@ directly caused a blank line preceded b backslash in a target when C++ was disabled. This confuses s versions of "make", apparently. Using an intermediate variab this. (Same for CPP_LOBJ.)
- 3. \$(LINK_FOR_BUILD) now includes \$(CFLAGS_FOR_BUILD) and \$(LINK) (non-Windows) now includes \$(CFLAGS) because these flags are som necessary on certain architectures.
- 4. Added a setting of -export-symbols-regex to the link command to those symbols that are exported in the C sense, but actually are within the library, and not documented. Their names all begin wi "_pcre_". This is not a perfect job, because (a) we have to exce symbols that pcretest ("illegally") uses, and (b) the facility i available (and never for static libraries). I have made a note t

find a way round (a) in the future.

Version 6.2 01-Aug-05

- There was no test for integer overflow of quantifier values. A c such as {111111111111111} would give undefined results. What is a minimum quantifier for a parenthesized subpattern overflowed a negative, the calculation of the memory size went wrong. This co led to memory overwriting.
- 2. Building PCRE using VPATH was broken. Hopefully it is now fixed.
- 3. Added "b" to the 2nd argument of fopen() in dftables.c, for nonoperating environments where this matters.
- 4. Applied Giuseppe Maxia's patch to add additional features for co PCRE options from within the C++ wrapper.
- 5. Named capturing subpatterns were not being correctly counted whe was compiled. This caused two problems: (a) If there were more t such subpatterns, the calculation of the memory needed for the w compiled pattern went wrong, leading to an overflow error. (b) N back references of the form \12, where the number was greater th not recognized as back references, even though there were suffic previous subpatterns.
- 6. Two minor patches to pcrecpp.cc in order to allow it to compile versions of gcc, e.g. 2.95.4.

Version 6.1 21-Jun-05

- 1. There was one reference to the variable "posix" in pcretest.c th surrounded by "#if !defined NOPOSIX".
- Make it possible to compile pcretest without DFA support, UTF8 s the cross-check on the old pcre_info() function, for the benefit cut-down version of PCRE that is currently imported into Exim.
- 3. A (silly) pattern starting with (?i)(?-i) caused an internal spa allocation error. I've done the easy fix, which wastes 2 bytes f patterns that start (?i) but I don't think that matters. The use just an example; this all applies to the other options as well.
- 4. Since libtool seems to echo the compile commands it is issuing, from "make" can be reduced a bit by putting "@" in front of each compile command.

- 5. Patch from the folks at Google for configure.in to be a bit more in checking for a suitable C++ installation before trying to com C++ stuff. This should fix a reported problem when a compiler wa but no suitable headers.
- 6. The man pages all had just "PCRE" as their title. I have changed be the relevant file name. I have also arranged that these names retained in the file doc/pcre.txt, which is a concatenation in t of all the man pages except the little individual ones for each
- 7. The NON-UNIX-USE file had not been updated for the different set files that come with release 6. I also added a few comments abou wrapper.

Version 6.0 07-Jun-05

- 1. Some minor internal re-organization to help with my DFA experime
- Some missing #ifdef SUPPORT_UCP conditionals in pcretest and pri didn't matter for the library itself when fully configured, but when compiling without UCP support, or within Exim, where the uc not imported.
- 3. Refactoring of the library code to split up the various function different source modules. The addition of the new DFA matching c below) to a single monolithic source would have made it really t unwieldy, quite apart from causing all the code to be include in statically linked application, when only some functions are used relevant even without the DFA addition now that patterns can be one application and matched in another.

The downside of splitting up is that there have to be some exter functions and data tables that are used internally in different the library but which are not part of the API. These have all ha names changed to start with "_pcre_" so that they are unlikely t with other external names.

- Added an alternate matching function, pcre_dfa_exec(), which mat a different (DFA) algorithm. Although it is slower than the orig function, it does have some advantages for certain types of matc problem.
- 5. Upgrades to pcretest in order to test the features of pcre_dfa_e including restarting after a partial match.
- 6. A patch for pcregrep that defines INVALID_FILE_ATTRIBUTES if it defined when compiling for Windows was sent to me. I have put it

code, though I have no means of testing or verifying it.

- 7. Added the pcre_refcount() auxiliary function.
- 8. Added the PCRE_FIRSTLINE option. This constrains an unanchored p match before or at the first newline in the subject string. In p the /f option on a pattern can be used to set this.
- 9. A repeated \w when used in UTF-8 mode with characters greater th would behave wrongly. This has been present in PCRE since releas
- 10. A number of changes to the pcregrep command:
 - (a) Refactored how -x works; insert ^(...)\$ instead of setting PCRE_ANCHORED and checking the length, in preparation for ad something similar for -w.
 - (b) Added the -w (match as a word) option.
 - (c) Refactored the way lines are read and buffered so as to have than one at a time available.
 - (d) Implemented a pcregrep test script.
 - (e) Added the -M (multiline match) option. This allows patterns over several lines of the subject. The buffering ensures tha 8K, or the rest of the document (whichever is the shorter) i for matching (and similarly the previous 8K for lookbehind a
 - (f) Changed the --help output so that it now says

-w, --word-regex(p)

instead of two lines, one with "regex" and the other with "r because that confused at least one person since the short fo same. (This required a bit of code, as the output is generat automatically from a table. It wasn't just a text change.)

- (g) -- can be used to terminate pcregrep options if the next thi option but starts with a hyphen. Could be a pattern or a pat starting with a hyphen, for instance.
- (h) "-" can be given as a file name to represent stdin.
- (i) When file names are being printed, "(standard input)" is use the standard input, for compatibility with GNU grep. Previou "<stdin>" was used.
- (j) The option --label=xxx can be used to supply a name to be us stdin when file names are being printed. There is no short f

- (k) Re-factored the options decoding logic because we are going two more options that take data. Such options can now be giv different ways, e.g. "-fname", "-f name", "--file=name", "--
- (1) Added the -A, -B, and -C options for requesting that lines o around matches be printed.
- (m) Added the -L option to print the names of files that do not any matching lines, that is, the complement of -l.
- (n) The return code is 2 if any file cannot be opened, but pcreg continue to scan other files.
- (o) The -s option was incorrectly implemented. For compatibility greps, it now suppresses the error message for a non-existen accessible file (but not the return code). There is a new op -q that suppresses the output of matching lines, which was w previously doing.
- (p) Added --include and --exclude options to specify files for i and exclusion when recursing.
- 11. The Makefile was not using the Autoconf-supported LDFLAGS macro Hopefully, it now does.
- 12. Missing cast in pcre_study().
- 13. Added an "uninstall" target to the makefile.
- 14. Replaced "extern" in the function prototypes in Makefile.in with "PCRE_DATA_SCOPE", which defaults to 'extern' or 'extern "C"' in world, but is set differently for Windows.
- 15. Added a second compiling function called pcre_compile2(). The on difference is that it has an extra argument, which is a pointer integer error code. When there is a compile-time failure, this i non-zero, in addition to the error test pointer being set to poi error message. The new argument may be NULL if no error number i (but then you may as well call pcre_compile(), which is now just wrapper). This facility is provided because some applications ne numeric error indication, but it has also enabled me to tidy up compile-time errors are handled in the POSIX wrapper.
- 16. Added VPATH=.libs to the makefile; this should help when buildin prefix path and installing with another. (Or so I'm told by some knows more about this stuff than I do.)
- 17. Added a new option, REG_DOTALL, to the POSIX function regcomp(). passes PCRE_DOTALL to the pcre_compile() function, making the ".

match everything, including newlines. This is not POSIX-compatib somebody wanted the feature. From pcretest it can be activated b both the P and the s flags.

- 18. AC_PROG_LIBTOOL appeared twice in Makefile.in. Removed one.
- 19. libpcre.pc was being incorrectly installed as executable.
- 20. A couple of places in pcretest check for end-of-line by looking it now also looks for '\r' so that it will work unmodified on Wi
- 21. Added Google's contributed C++ wrapper to the distribution.
- 22. Added some untidy missing memory free() calls in pcretest, to ke Electric Fence happy when testing.

Version 5.0 13-Sep-04

- 1. Internal change: literal characters are no longer packed up into containing multiple characters in a single byte-string. Each cha is now matched using a separate opcode. However, there may be mo byte in the character in UTF-8 mode.
- 2. The pcre_callout_block structure has two new fields: pattern_pos next_item_length. These contain the offset in the pattern to the item, and its length, respectively.
- 3. The PCRE_AUTO_CALLOUT option for pcre_compile() requests the aut insertion of callouts before each pattern item. Added the /C opt pcretest to make use of this.
- 4. On the advice of a Windows user, the lines

#if defined(_WIN32) || defined(WIN32)
_setmode(_fileno(stdout), 0x8000);
#endif /* defined(_WIN32) || defined(WIN32) */

have been added to the source of pcretest. This apparently does magic in relation to line terminators.

- 5. Changed "r" and "w" in the calls to fopen() in pcretest to "rb" for the benefit of those environments where the "b" makes a diff
- 6. The icc compiler has the same options as gcc, but "configure" do to know about it. I have put a hack into configure.in that adds to set GCC=yes if CC=icc. This seems to end up at a point in the generated configure script that is early enough to affect the se

compiler options, which is what is needed, but I have no means c whether it really works. (The user who reported this had patched generated configure script, which of course I cannot do.)

LATER: After change 22 below (new libtool files), the configure seems to know about icc (and also ecc). Therefore, I have commen this hack in configure.in.

- 7. Added support for pkg-config (2 patches were sent in).
- 8. Negated POSIX character classes that used a combination of inter were completely broken. These were [[:^alpha:]], [[:^alnum:]], a [[:^ascii]]. Typically, they would match almost any characters. POSIX classes were not broken in this way.
- 9. Matching the pattern "\b.*?" against "ab cd", starting at offset to find the match, as PCRE was deluded into thinking that the ma start at the start point or following a newline. The same bug ap patterns with negative forward assertions or any backward assert preceding ".*" at the start, unless the pattern required a fixed character. This was a failing pattern: "(?!.bcd).*". The bug is
- 10. In UTF-8 mode, when moving forwards in the subject after a faile starting at the last subject character, bytes beyond the end of string were read.
- 11. Renamed the variable "class" as "classbits" to make life easier users. (Previously there was a macro definition, but it apparent enough.)
- 12. Added the new field "tables" to the extra data so that tables ca in at exec time, or the internal tables can be re-selected. This a compiled regex to be saved and re-used at a later time by a di program that might have everything at different addresses.
- 13. Modified the pcre-config script so that, when run on Solaris, it -R library as well as a -L library.
- 14. The debugging options of pcretest (-d on the command line or D c pattern) showed incorrect output for anything following an exten that contained multibyte characters and which was followed by a
- 15. Added optional support for general category Unicode character pr via the \p, \P, and \X escapes. Unicode property support implies support. It adds about 90K to the size of the library. The meani inbuilt class escapes such as \d and \s have NOT been changed.
- 16. Updated pcredemo.c to include calls to free() to release the mem compiled pattern.

- 17. The generated file chartables.c was being created in the source instead of in the building directory. This caused the build to f source directory was different from the building directory, and read-only.
- 18. Added some sample Win commands from Mark Tetrode into the NON-UN file. No doubt somebody will tell me if they don't make sense... Dan Mooney's comments about building on OpenVMS.
- 19. Added support for partial matching via the PCRE_PARTIAL option f pcre_exec() and the \P data escape in pcretest.
- 20. Extended pcretest with 3 new pattern features:
 - (i) A pattern option of the form ">rest-of-line" causes pcrete write the compiled pattern to the file whose name is "rest This is a straight binary dump of the data, with the saved the character tables forced to be NULL. The study data, if written too. After writing, pcretest reads a new pattern.
 - (ii) If, instead of a pattern, "<rest-of-line" is given, pcrete compiled pattern from the given file. There must not be an occurrences of "<" in the file name (pretty unlikely); if pcretest will instead treat the initial "<" as a pattern d After reading in the pattern, pcretest goes on to read dat usual.
 - (iii) The F pattern option causes pcretest to flip the bytes in and 16-bit fields in a compiled pattern, to simulate a pat was compiled on a host of opposite endianness.
- 21. The pcre-exec() function can now cope with patterns that were co hosts of opposite endianness, with this restriction:

As for any compiled expression that is saved and used later, t pointer field cannot be preserved; the extra_data field in the to pcre_exec() should be used to pass in a tables address if a other than the default internal tables were used at compile ti

- 22. Calling pcre_exec() with a negative value of the "ovecsize" para now diagnosed as an error. Previously, most of the time, a negat would have been treated as zero, but if in addition "ovector" wa NULL, a crash could occur.
- 23. Updated the files ltmain.sh, config.sub, config.guess, and acloc new versions from the libtool 1.5 distribution (the last one is a file called libtool.m4). This seems to have fixed the need to "configure" to support Darwin 1.3 (which I used to do). However, had to patch ltmain.sh to ensure that \${SED} is set (it isn't on workstation).

24. Changed the PCRE licence to be the more standard "BSD" licence.

Version 4.5 01-Dec-03

1. There has been some re-arrangement of the code for the match() f that it can be compiled in a version that does not call itself r Instead, it keeps those local variables that need separate insta each "recursion" in a frame on the heap, and gets/frees frames w needs to "recurse". Keeping track of where control must go is do of setjmp/longjmp. The whole thing is implemented by a set of ma hide most of the details from the main code, and operates only i NO_RECURSE is defined while compiling pcre.c. If PCRE is built u "configure" mechanism, "--disable-stack-for-recursion" turns on operating.

To make it easier for callers to provide specially tailored get/ functions for this usage, two new functions, pcre_stack_malloc, pcre_stack_free, are used. They are always called in strict stac and the size of block requested is always the same.

The PCRE_CONFIG_STACKRECURSE info parameter can be used to find PCRE has been compiled to use the stack or the heap for recursic -C option of pcretest uses this to show which version is compile

A new data escape \S , is added to pcretest; it causes the amount obtained and freed by both kinds of malloc/free at match time to to the output.

- 2. Changed the locale test to use "fr_FR" instead of "fr" because t what's available on my current Linux desktop machine.
- 3. When matching a UTF-8 string, the test for a valid string at the been extended. If start_offset is not zero, PCRE now checks that to a byte that is the start of a UTF-8 character. If not, it ret PCRE_ERROR_BADUTF8_OFFSET (-11). Note: the whole string is still this is necessary because there may be backward assertions in th When matching the same subject several times, it may save resour PCRE_NO_UTF8_CHECK on all but the first call if the string is lo
- 4. The code for checking the validity of UTF-8 strings has been tig that it rejects (a) strings containing 0xfe or 0xff bytes and (b containing "overlong sequences".
- 5. Fixed a bug (appearing twice) that I could not find any way of e I had written "if ((digitab[*p++] && chtab_digit) == 0)" where t should have been "&", but it just so happened that all the cases through by mistake were picked up later in the function.

- 6. I had used a variable called "isblank" this is a C99 function, some compilers to warn. To avoid this, I renamed it (as "blankcl
- 7. Cosmetic: (a) only output another newline at the end of pcretest prompting; (b) run "./pcretest /dev/null" at the start of the te so the version is shown; (c) stop "make test" echoing "./RunTest
- 8. Added patches from David Burgess to enable PCRE to run on EBCDIC
- 9. The prototype for memmove() for systems that don't have it was u size_t, but the inclusion of the header that defines size_t was moved the #includes for the C headers earlier to avoid this.
- 10. Added some adjustments to the code to make it easier to compiler special systems:
 - (a) Some "const" qualifiers were missing.
 - (b) Added the macro EXPORT before all exported functions; by d is defined to be empty.
 - (c) Changed the dftables auxiliary program (that builds charta that it reads its output file name as an argument instead to the standard output and assuming this can be redirected
- 11. In UTF-8 mode, if a recursive reference (e.g. (?1)) followed a c class containing characters with values greater than 255, PCRE c went into a loop.
- 12. A recursive reference to a subpattern that was within another su that had a minimum quantifier of zero caused PCRE to crash. For (x(y(?2))z)? provoked this bug with a subject that got as far as recursion. If the recursively-called subpattern itself had a zer that was OK.
- 13. In pcretest, the buffer for reading a data line was set at 30K, buffer into which it was copied (for escape processing) was stil 1024, so long lines caused crashes.
- 14. A pattern such as /[ab]{1,3}+/ failed to compile, giving the err "internal error: code overflow...". This applied to any characte that was followed by a possessive quantifier.
- 15. Modified the Makefile to add libpcre.la as a prerequisite for libpcreposix.la because I was told this is needed for a parallel work.
- 16. If a pattern that contained .* following optional items at the s studied, the wrong optimizing data was generated, leading to mat errors. For example, studying /[ab]*.*c/ concluded, erroneously, matching string must start with a or b or c. The correct conclus

this pattern is that a match can start with any character.

Version 4.4 13-Aug-03

- 1. In UTF-8 mode, a character class containing characters with valu 127 and 255 was not handled correctly if the compiled pattern wa In fixing this, I have also improved the studying algorithm for classes (slightly).
- 2. Three internal functions had redundant arguments passed to them. might give a very teeny performance improvement.
- 3. Documentation bug: the value of the capture_top field in a callo more than* the number of the hightest numbered captured substrin
- 4. The Makefile linked pcretest and pcregrep with -lpcre, which cou in incorrectly linking with a previously installed version. They explicitly with libpcre.la.
- 5. configure.in no longer needs to recognize Cygwin specially.
- 6. A problem in pcre.in for Windows platforms is fixed.
- 7. If a pattern was successfully studied, and the -d (or /D) flag w pcretest, it used to include the size of the study block as part output. Unfortunately, the structure contains a field that has a size on different hardware architectures. This meant that the te showed this size failed. As the block is currently always of a f this information isn't actually particularly useful in pcretest I have just removed it.
- 8. Three pre-processor statements accidentally did not start in col Sadly, there are *still* compilers around that complain, even th standard C has not required this for well over a decade. Sigh.
- 9. In pcretest, the code for checking callouts passed small integer callout_data field, which is a void * field. However, some picky complained about the casts involved for this on 64-bit systems. pcretest passes the address of the small integer instead, which rid of the warnings.
- 10. By default, when in UTF-8 mode, PCRE now checks for valid UTF-8 both compile and run time, and gives an error if an invalid UTFis found. There is a option for disabling this check in cases wh string is known to be correct and/or the maximum performance is
- 11. In response to a bug report, I changed one line in Makefile.in f

-Wl,--out-implib,.libs/lib@WIN_PREFIX@pcreposix.dll.a \
to
 -Wl,--out-implib,.libs/@WIN_PREFIX@libpcreposix.dll.a \
to look similar to other lines, but I have no way of telling whe
is the right thing to do, as I do not use Windows. No doubt I'll
if it's wrong...

Version 4.3 21-May-03

- Two instances of @WIN_PREFIX@ omitted from the Windows targets in Makefile.
- 2. Some refactoring to improve the quality of the code:
 - (i) The utf8_table... variables are now declared "const".
 - (ii) The code for \cx, which used the "case flipping" table to u lower case letters, now just substracts 32. This is ASCII-s but the whole concept of \cx is ASCII-specific, so it seems reasonable.
 - (iii) PCRE was using its character types table to recognize decim hexadecimal digits in the pattern. This is silly, because i only 0-9, a-f, and A-F, but the character types table is lo specific, which means strange things might happen. A privat table is now used for this - though it costs 256 bytes, a t much faster than multiple explicit tests. Of course, the st character types table is still used for matching digits in strings against \d.
 - (iv) Strictly, the identifier ESC_t is reserved by POSIX (all id ending in _t are). So I've renamed it as ESC_tee.
- 3. The first argument for regexec() in the POSIX wrapper should have defined as "const".
- 4. Changed pcretest to use malloc() for its buffers so that they can Electric Fenced for debugging.
- 5. There were several places in the code where, in UTF-8 mode, PCRE to read one or more bytes before the start of the subject string. had no effect on PCRE's behaviour, but in some circumstances it c provoke a segmentation fault.
- 6. A lookbehind at the start of a pattern in UTF-8 mode could also c to try to read one or more bytes before the start of the subject

7. A lookbehind in a pattern matched in non-UTF-8 mode on a PCRE com UTF-8 support could misbehave in various ways if the subject stri contained bytes with the 0x80 bit set and the 0x40 bit unset in a area. (PCRE was not checking for the UTF-8 mode flag, and trying back over UTF-8 characters.)

Version 4.2 14-Apr-03

- 1. Typo "#if SUPPORT_UTF8" instead of "#ifdef SUPPORT_UTF8" fixed.
- 2. Changes to the building process, supplied by Ronald Landheer-Cies [ON_WINDOWS]: new variable, "#" on non-Windows platforms [NOT_ON_WINDOWS]: new variable, "#" on Windows platforms [WIN_PREFIX]: new variable, "cyg" for Cygwin * Makefile.in: use autoconf substitution for OBJEXT, EXEEXT, BU and BUILD EXEEXT Note: automatic setting of the BUILD variables is not yet worki set CPPFLAGS and BUILD CPPFLAGS (but don't use yet) - should be compile-time but not at link-time [LINK]: use for linking executables only make different versions for Windows and non-Windows [LINKLIB]: new variable, copy of UNIX-style LINK, used for link libraries [LINK_FOR_BUILD]: new variable [OBJEXT]: use throughout [EXEEXT]: use throughout <winshared>: new target <wininstall>: new target <dftables.o>: use native compiler <dftables>: use native linker <install>: handle Windows platform correctly <clean>: ditto <check>: ditto copy DLL to top builddir before testing

As part of these changes, -no-undefined was removed again. This w to give trouble on HP-UX 11.0, so getting rid of it seems like a in any case.

3. Some tidies to get rid of compiler warnings:

- . In the match_data structure, match_limit was an unsigned long i match_call_count was an int. I've made them both unsigned long
- . In pcretest the fact that a const uschar * doesn't automaticall a void * provoked a warning.
- . Turning on some more compiler warnings threw up some "shadow" v

and a few more missing casts.

- 4. If PCRE was complied with UTF-8 support, but called without the P option, a class that contained a single character with a value be and 255 (e.g. /[\xFF]/) caused PCRE to crash.
- 5. If PCRE was compiled with UTF-8 support, but called without the P option, a class that contained several characters, but with at le whose value was between 128 and 255 caused PCRE to crash.

Version 4.1 12-Mar-03

1. Compiling with gcc -pedantic found a couple of places where casts needed, and a string in dftables.c that was longer than standard com required to support.

2. Compiling with Sun's compiler found a few more places where the c be tidied up in order to avoid warnings.

3. The variables for cross-compiling were called HOST_CC and HOST_CF first of these names is deprecated in the latest Autoconf in favour CC_FOR_BUILD, because "host" is typically used to mean the system on compiled code will be run. I can't find a reference for HOST_CFLAGS, analogy I have changed it to CFLAGS_FOR_BUILD.

4. Added -no-undefined to the linking command in the Makefile, becau apparently helpful for Windows. To make it work, also added "-L. -lp linking step for the pcreposix library.

5. PCRE was failing to diagnose the case of two named groups with th name.

6. A problem with one of PCRE's optimizations was discovered. PCRE r literal character that is needed in the subject for a match, and sca ensure that it is present before embarking on the full matching proc saves time in cases of nested unlimited repeats that are never going Problem: the scan can take a lot of time if the subject is very long megabytes), thus penalizing straightforward matches. It is now done amount of subject to be scanned is less than 1000 bytes.

7. A lesser problem with the same optimization is that it was record first character of an anchored pattern as "needed", thus provoking a right along the subject, even when the first match of the pattern wa fail. The "needed" character is now not set for anchored patterns, u follows something in the pattern that is of non-fixed length. Thus, fulfils its original purpose of finding quick non-matches in cases c unlimited repeats, but isn't used for simple anchored patterns such Version 4.0 17-Feb-03

1. If a comment in an extended regex that started immediately after extended to the end of string, PCRE compiled incorrect data. This cc all kinds of weird effects. Example: /#/ was bad; /()#/ was bad; /a#

2. Moved to autoconf 2.53 and libtool 1.4.2.

3. Perl 5.8 no longer needs "use utf8" for doing UTF-8 things. Conse the special perltest8 script is no longer needed - all the tests can from a single perltest script.

4. From 5.004, Perl has not included the VT character (0x0b) in the by s. It has now been removed in PCRE. This means it isn't recogniz whitespace in /x regexes too, which is the same as Perl. Note that t class [:space:] *does* include VT, thereby creating a mess.

5. Added the class [:blank:] (a GNU extension from Perl 5.8) to matc space and tab.

6. Perl 5.005 was a long time ago. It's time to amalgamate the tests its new features into the main test script, reducing the number of s

7. Perl 5.8 has changed the meaning of patterns like /a(?i)b/. Earli were backward compatible, and made the (?i) apply to the whole patte /i were given. Now it behaves more logically, and applies the option only to what follows. PCRE has been changed to follow suit. However, finds options settings right at the start of the pattern, it extract the global options, as before. Thus, they show up in the info data.

8. Added support for the \Q...\E escape sequence. Characters in betw treated as literals. This is slightly different from Perl in that \$ also handled as literals inside the quotes. In Perl, they will cause interpolation. Note the following examples:

Pattern	PCRE matches	Perl matches
\Qabc\$xyz\E	abc\$xyz	abc followed by the content
\Qabc\\$xyz\E	abc\\$xyz	abc\\$xyz
\Qabc\E\\$\Qxyz\E	abc\$xyz	abc\$xyz

For compatibility with Perl, Q...E sequences are recognized inside classes as well as outside them.

9. Re-organized 3 code statements in pcretest to avoid "overflow in floating-point constant arithmetic" warnings from a Microsoft compil (size_t) cast to one statement in pcretest and one in pcreposix to a signed/unsigned warnings.

10. SunOS4 doesn't have strtoul(). This was used only for unpicking option for pcretest, so I've replaced it by a simple function that d that job.

11. pcregrep was ending with code 0 instead of 2 for the commands "p "pcregrep -".

12. Added "possessive quantifiers" ?+, *+, ++, and $\{,\}$ + which come f Java package. This provides some syntactic sugar for simple cases of documentation calls "once-only subpatterns". A pattern such as x*+ i as (?>x*). In other words, if what is inside (?>...) is just a singl item, you can use this simplified notation. Note that only makes sen greedy quantifiers. Consequently, the use of the possessive quantifi greediness, whatever the setting of the PCRE_UNGREEDY option.

13. A change of greediness default within a pattern was not taking e the current level for patterns like /(b+(?U)a+)/. It did apply to pa subpatterns that followed. Patterns like /b+(?U)a+/ worked because t was abstracted outside.

14. PCRE now supports the G assertion. It is true when the current position is at the start point of the match. This differs from A wh starting offset is non-zero. Used with the /g option of pcretest (or code), it works in the same way as it does for Perl's /g option. If alternatives of a regex begin with G, the expression is anchored to match position, and the "anchored" flag is set in the compiled expre

15. Some bugs concerning the handling of certain option changes with have been fixed. These applied to options other than (?ims). For exa "a(?x: b c)d" did not match "XabcdY" but did match "Xa b c dY". It been the other way round. Some of this was related to change 7 above

16. PCRE now gives errors for /[.x.]/ and /[=x=]/ as unsupported POS features, as Perl does. Previously, PCRE gave the warnings only for and /[[=x=]]/. PCRE now also gives an error for /[:name:]/ because i POSIX classes only within a class (e.g. /[[:alpha:]]/).

17. Added support for Perl's C escape. This matches one byte, even mode. Unlike ".", it always matches newline, whatever the setting of PCRE_DOTALL. However, PCRE does not permit C to appear in lookbehin assertions. Perl allows it, but it doesn't (in general) work because calculate the length of the lookbehind. At least, that's the case fo 5.8.0 - I've been told they are going to document that it doesn't wo future.

18. Added an error diagnosis for escapes that PCRE does not support: L, l, N, P, p, U, u, and X.

19. Although correctly diagnosing a missing ']' in a character class

reading past the end of the pattern in cases such as /[abcd/.

20. PCRE was getting more memory than necessary for patterns with cl contained both POSIX named classes and other characters, e.g. /[[:sp

21. Added some code, conditional on #ifdef VPCOMPAT, to make life ea compiling PCRE for use with Virtual Pascal.

22. Small fix to the Makefile to make it work properly if the build outside the source tree.

23. Added a new extension: a condition to go with recursion. If a co subpattern starts with (?(R) the "true" branch is used if recursion happened, whereas the "false" branch is used only at the top level.

24. When there was a very long string of literal characters (over 25 without UTF support, over 250 bytes with UTF support), the computati much memory was required could be incorrect, leading to segfaults or strange effects.

25. PCRE was incorrectly assuming anchoring (either to start of subj start of line for a non-DOTALL pattern) when a pattern started with there was a subsequent back reference to those brackets. This meant example, $/(.*)\d+\1/$ failed to match "abc123bc". Unfortunately, it i possible to check for precisely this case. All we can do is abandon optimization if .* occurs inside capturing brackets when there are a references whatsoever. (See below for a better fix that came later.)

26. The handling of the optimization for finding the first character non-anchored pattern, and for finding a character that is required 1 match were failing in some cases. This didn't break the matching; it failed to optimize when it could. The way this is done has been re-i

27. Fixed typo in error message for invalid (?R item (it said "(?p"))

28. Added a new feature that provides some of the functionality that provides with (?{...}). The facility is termed a "callout". The way in PCRE is for the caller to provide an optional function, by settin pcre_callout to its entry point. Like pcre_malloc and pcre_free, thi global variable. By default it is unset, which disables all calling the function called, the regex must include (?C) at appropriate poin is, in fact, equivalent to (?C0), and any number <= 255 may be given This provides a means of identifying different callout points. When reaches such a point in the regex, if pcre_callout has been set, the function is called. It is provided with data in a structure called pcre_callout_block, which is defined in pcre.h. If the function retu matching continues; if it returns a non-zero value, the match at the point fails. However, backtracking will occur if possible. [This was later and other features added - see item 49 below.] 29. pcretest is upgraded to test the callout functionality. It provi callout function that displays information. By default, it shows the the match and the current position in the text. There are some new d to vary what happens:

\C+	in addition, show current contents of captured subst
\C-	do not supply a callout function
\C!n	return 1 when callout number n is reached
\C!n!m	return 1 when callout number n is reached for the mt

30. If pcregrep was called with the -l option and just a single file output "<stdin>" if a match was found, instead of the file name.

31. Improve the efficiency of the POSIX API to PCRE. If the number o slots is less than POSIX_MALLOC_THRESHOLD, use a block on the stack pcre_exec(). This saves a malloc/free per call. The default value of POSIX_MALLOC_THRESHOLD is 10; it can be changed by --with-posix-mall when configuring.

32. The default maximum size of a compiled pattern is 64K. There hav few cases of people hitting this limit. The code now uses macros to storing of links as offsets within the compiled pattern. It defaults links, but this can be changed to 3 or 4 bytes by --with-link-size w configuring. Tests 2 and 5 work only with 2-byte links because they debugging information about compiled patterns.

33. Internal code re-arrangements:

- (a) Moved the debugging function for printing out a compiled regex i its own source file (printint.c) and used #include to pull it in pcretest.c and, when DEBUG is defined, into pcre.c, instead of h separate copies.
- (b) Defined the list of op-code names for debugging as a macro in internal.h so that it is next to the definition of the opcodes.
- (c) Defined a table of op-code lengths for simpler skipping along co code. This is again a macro in internal.h so that it is next to definition of the opcodes.

34. Added support for recursive calls to individual subpatterns, alo lines of Robin Houston's patch (but implemented somewhat differently

35. Further mods to the Makefile to help Win32. Also, added code to allow it to read and process whole directories in Win32. This code w contributed by Lionel Fourquaux; it has not been tested by me.

36. Added support for named subpatterns. The Python syntax (?P<name> used to name a group. Names consist of alphanumerics and underscores be unique. Back references use the syntax (?P=name) and recursive ca

(?P>name) which is a PCRE extension to the Python extension. Groups numbers. The function pcre_fullinfo() can be used after compilation a name/number map. There are three relevant calls:

PCRE_INFO_NAMEENTRYSIZE	yields the size of each entry in th
PCRE_INFO_NAMECOUNT	yields the number of entries
PCRE_INFO_NAMETABLE	yields a pointer to the map.

The map is a vector of fixed-size entries. The size of each entry de the length of the longest name used. The first two bytes of each ent group number, most significant byte first. There follows the corresp name, zero terminated. The names are in alphabetical order.

37. Make the maximum literal string in the compiled code 250 for the case instead of 255. Making it the same both with and without UTF-8 means that the same test output works with both.

38. There was a case of malloc(0) in the POSIX testing code in pcret calling malloc() with a zero argument.

39. Change 25 above had to resort to a heavy-handed test for the .* optimization. I've improved things by keeping a bitmap of backrefere numbers 1-31 so that if .* occurs inside capturing brackets that are fact referenced, the optimization can be applied. It is unlikely tha relevant occurrence of .* (i.e. one which might indicate anchoring o the match to follow n) will appear inside brackets with a number gr 31, but if it does, any back reference > 31 suppresses the optimizat

40. Added a new compile-time option PCRE_NO_AUTO_CAPTURE. This has t of disabling numbered capturing parentheses. Any opening parenthesis not followed by ? behaves as if it were followed by ?: but named par can still be used for capturing (and they will acquire numbers in th way).

41. Redesigned the return codes from the match() function into yes/n that errors can be passed back from deep inside the nested calls. A failure while inside a recursive subpattern call now causes the PCRE_ERROR_NOMEMORY return instead of quietly going wrong.

42. It is now possible to set a limit on the number of times the mat function is called in a call to pcre_exec(). This facility makes it limit the amount of recursion and backtracking, though not in a dire obvious way, because the match() function is used in a number of dif circumstances. The count starts from zero for each position in the s string (for non-anchored patterns). The default limit is, for compat large number, namely 10 000 000. You can change this in two ways:

(a) When configuring PCRE before making, you can use --with-match-li to set a default value for the compiled library. (b) For each call to pcre_exec(), you can pass a pcre_extra block in a different value is set. See 45 below.

If the limit is exceeded, pcre_exec() returns PCRE_ERROR_MATCHLIMIT.

43. Added a new function pcre_config(int, void *) to enable run-time of things that can be changed at compile time. The first argument sp what is wanted and the second points to where the information is to The current list of available information is:

PCRE_CONFIG_UTF8

The output is an integer that is set to one if UTF-8 support is avai otherwise it is set to zero.

PCRE_CONFIG_NEWLINE

The output is an integer that it set to the value of the code that i newline. It is either LF (10) or CR (13).

PCRE_CONFIG_LINK_SIZE

The output is an integer that contains the number of bytes used for linkage in compiled expressions. The value is 2, 3, or 4. See item 3

PCRE_CONFIG_POSIX_MALLOC_THRESHOLD

The output is an integer that contains the threshold above which the interface uses malloc() for output vectors. See item 31 above.

PCRE_CONFIG_MATCH_LIMIT

The output is an unsigned integer that contains the default limit of of match() calls in a pcre_exec() execution. See 42 above.

44. pcretest has been upgraded by the addition of the -C option. Thi to extract all the available output from the new pcre_config() funct output it. The program then exits immediately.

45. A need has arisen to pass over additional data with calls to pcr order to support additional features. One way would have been to def pcre_exec2() (for example) with extra arguments, but this would not extensible, and would also have required all calls to the original f be mapped to the new one. Instead, I have chosen to extend the mecha is used for passing in "extra" data from pcre_study().

The pcre_extra structure is now exposed and defined in pcre.h. It cu contains the following fields:

flags a bitmap indicating which of the following fields ar

study_data	opaque data from pcre_study()
match_limit	a way of specifying a limit on match() calls for a s
	call to pcre_exec()
callout data	data for callouts (see 49 below)

The flag bits are also defined in pcre.h, and are

PCRE_EXTRA_STUDY_DATA PCRE_EXTRA_MATCH_LIMIT PCRE_EXTRA_CALLOUT_DATA

The pcre_study() function now returns one of these new pcre_extra bl the actual study data pointed to by the study_data field, and the PCRE_EXTRA_STUDY_DATA flag set. This can be passed directly to pcre_ before. That is, this change is entirely upwards-compatible and requ change to existing code.

If you want to pass in additional data to pcre_exec(), you can eithe in a pcre_extra block provided by pcre_study(), or create your own p block.

46. pcretest has been extended to test the PCRE_EXTRA_MATCH_LIMIT fe data string contains the escape sequence \M , pcretest calls pcre_exe times with different match limits, until it finds the minimum value pcre_exec() to complete. The value is then output. This can be instr most simple matches the number is quite small, but for pathological gets very large very quickly.

47. There's a new option for pcre_fullinfo() called PCRE_INFO_STUDYS returns the size of the data block pointed to by the study_data fiel pcre_extra block, that is, the value that was passed as the argument pcre_malloc() when PCRE was getting memory in which to place the inf created by pcre_study(). The fourth argument should point to a size_pcretest has been extended so that this information is shown after a pcre_study() call when information about the compiled regex is being

48. Cosmetic change to Makefile: there's no need to have / after \$(D because what follows is always an absolute path. (Later: it turns ou is more than cosmetic for MinGW, because it doesn't like empty path components.)

49. Some changes have been made to the callout feature (see 28 above

(i) A callout function now has three choices for what it returns:

0	=>	success, carry on matching
> 0	=>	failure at this point, but backtrack if possible
< 0	=>	serious error, return this value from pcre_exec()

Negative values should normally be chosen from the set of PCRE_

values. In particular, returning PCRE_ERROR_NOMATCH forces a st "match failed" error. The error number PCRE_ERROR_CALLOUT is re use by callout functions. It will never be used by PCRE itself.

- (ii) The pcre_extra structure (see 45 above) has a void * field call callout_data, with corresponding flag bit PCRE_EXTRA_CALLOUT_DA pcre_callout_block structure has a field of the same name. The the field passed in the pcre_extra structure are passed to the function in the corresponding field in the callout block. This easier to use the same callout-containing regex from multiple t testing, the pcretest program has a new data escape
 - \C*n pass the number n (may be negative) as callout_da

If the callout function in pcretest receives a non-zero value a callout_data, it returns that value.

50. Makefile wasn't handling CFLAGS properly when compiling dftables there were some redundant \$(CFLAGS) in commands that are now specifi \$(LINK), which already includes \$(CFLAGS).

51. Extensions to UTF-8 support are listed below. These all apply wh has been compiled with UTF-8 support *and* pcre_compile() has been c with the PCRE_UTF8 flag. Patterns that are compiled without that fla one-byte characters throughout. Note that case-insensitive matching only to characters whose values are less than 256. PCRE doesn't supp notion of cases for higher-valued characters.

- (i) A character class whose characters are all within 0-255 is han a bit map, and the map is inverted for negative classes. Previ character > 255 always failed to match such a class; however i match if the class was a negative one (e.g. [^ab]). This has b
- (ii) A negated character class with a single character < 255 is cod "not this character" (OP_NOT). This wasn't working properly wh character was multibyte, either singly or repeated.
- (iii) Repeats of multibyte characters are now handled correctly in U mode, for example: $x{100}{2,3}$.
- (iv) The character escapes \b, \B, \d, \D, \s, \S, \w, and \W (eith singly or repeated) now correctly test multibyte characters. H PCRE doesn't recognize any characters with values greater than digits, spaces, or word characters. Such characters always mat and \W, and never match \d, \s, or \w.
- (v) Classes may now contain characters and character ranges with v greater than 255. For example: $[ab\x{100}-\x{400}]$.
- (vi) pcregrep now has a --utf-8 option (synonym -u) which makes it

PCRE in UTF-8 mode.

52. The info request value PCRE_INFO_FIRSTCHAR has been renamed PCRE_INFO_FIRSTBYTE because it is a byte value. However, the old nam retained for backwards compatibility. (Note that LASTLITERAL is also value.)

53. The single man page has become too large. I have therefore split a number of separate man pages. These also give rise to individual H these are now put in a separate directory, and there is an index.htm lists them all. Some hyperlinking between the pages has been install

54. Added convenience functions for handling named capturing parenth

55. Unknown escapes inside character classes (e.g. [M]) and escapes aren't interpreted therein (e.g. [C]) are literals in Perl. This is true in PCRE, except when the PCRE_EXTENDED option is set, in which are faulted.

56. Introduced HOST_CC and HOST_CFLAGS which can be set in the envir calling configure. These values are used when compiling the dftables which is run to generate the source of the default character tables. default to the values of CC and CFLAGS. If you are cross-compiling P you will need to set these values.

57. Updated the building process for Windows DLL, as provided by Fre

Version 3.9 02-Jan-02

1. A bit of extraneous text had somehow crept into the pcregrep docu

2. If --disable-static was given, the building process failed when t build pcretest and pcregrep. (For some reason it was using libtool t them, which is not right, as they aren't part of the library.)

Version 3.8 18-Dec-01

1. The experimental UTF-8 code was completely screwed up. It was pac bytes in the wrong order. How dumb can you get?

Version 3.7 29-Oct-01

1. In updating pcretest to check change 1 of version 3.6, I screwed This caused pcretest, when used on the test data, to segfault. Unfor this didn't happen under Solaris 8, where I normally test things.

2. The Makefile had to be changed to make it work on BSD systems, wh doesn't seem to recognize that ./xxx and xxx are the same file. (Thi isn't in ChangeLog distributed with 3.7 because I forgot when I hast this fix an hour or so after the initial 3.7 release.)

Version 3.6 23-Oct-01

1. Crashed with /(sens|respons)e and \1ibility/ and "sense and sensi offsets passed as NULL with zero offset count.

2. The config.guess and config.sub files had not been updated when I the latest autoconf.

Version 3.5 15-Aug-01

1. Added some missing #if !defined NOPOSIX conditionals in pcretest. had been forgotten.

2. By using declared but undefined structures, we can avoid using "v definitions in pcre.h while keeping the internal definitions of the private.

3. The distribution is now built using autoconf 2.50 and libtool 1.4 user point of view, this means that both static and shared libraries by default, but this can be individually controlled. More of the wor handling this static/shared cases is now inside libtool instead of P file.

4. The pcretest utility is now installed along with pcregrep because useful for users (to test regexs) and by doing this, it automaticall relinked by libtool. The documentation has been turned into a man pa there are now .1, .txt, and .html versions in /doc.

5. Upgrades to pcregrep:

(i) Added long-form option names like gnu grep.

(ii) Added --help to list all options with an explanatory phrase

(iii) Added -r, --recursive to recurse into sub-directories.

(iv) Added -f, --file to read patterns from a file.

6. pcre_exec() was referring to its "code" argument before testing t argument for NULL (and giving an error if it was NULL).

7. Upgraded Makefile.in to allow for compiling in a different direct the source directory.

8. Tiny buglet in pcretest: when pcre_fullinfo() was called to retri options bits, the pointer it was passed was to an int instead of to long int. This mattered only on 64-bit systems.

9. Fixed typo (3.4/1) in pcre.h again. Sigh. I had changed pcre.h (w generated) instead of pcre.in, which it its source. Also made the sa in several of the .c files.

10. A new release of gcc defines printf() as a macro, which broke pc because it had an ifdef in the middle of a string argument for print by using separate calls to printf().

11. Added --enable-newline-is-cr and --enable-newline-is-lf to the c script, to force use of CR or LF instead of \n in the source. On non systems, the value can be set in config.h.

12. The limit of 200 on non-capturing parentheses is a _nesting_ lim absolute limit. Changed the text of the error message to make this c likewise updated the man page.

13. The limit of 99 on the number of capturing subpatterns has been The new limit is 65535, which I hope will not be a "real" limit.

Version 3.4 22-Aug-00

1. Fixed typo in pcre.h: unsigned const char * changed to const unsi

2. Diagnose condition (?(0) as an error instead of crashing on match

Version 3.3 01-Aug-00

1. If an octal character was given, but the value was greater than $\$ was not getting masked to the least significant bits, as documented. lead to crashes in some systems.

2. Perl 5.6 (if not earlier versions) accepts classes like [a-\d] an the hyphen as a literal. PCRE used to give an error; it now behaves

3. Added the functions pcre_free_substring() and pcre_free_substring These just pass their arguments on to (pcre_free)(), but they are pr because some uses of PCRE bind it to non-C systems that can call its but cannot call free() or pcre_free() directly.

4. Add "make test" as a synonym for "make check". Corrected some com the Makefile.

5. Add \$(DESTDIR)/ in front of all the paths in the "install" target Makefile.

6. Changed the name of pgrep to pcregrep, because Solaris has introd command called pgrep for grepping around the active processes.

7. Added the beginnings of support for UTF-8 character strings.

8. Arranged for the Makefile to pass over the settings of CC, CFLAGS RANLIB to ./ltconfig so that they are used by libtool. I think these the relevant ones. (AR is not passed because ./ltconfig does its own out for the ar command.)

Version 3.2 12-May-00

This is purely a bug fixing release.

1. If the pattern /((Z)+|A)*/ was matched agained ZABCDEFG it matche of ZA. This was just one example of several cases that could provoke which was introduced by change 9 of version 2.00. The code for break infinite loops after an iteration that matches an empty string was't correctly.

2. The pcretest program was not imitating Perl correctly for the pat when matched against abbab (for example). After matching an empty st wasn't forcing anchoring when setting PCRE_NOTEMPTY for the next att caused it to match further down the string than it should.

3. The code contained an inclusion of sys/types.h. It isn't clear wh was there because it doesn't seem to be needed, and it causes troubl systems, as it is not a Standard C header. It has been removed.

4. Made 4 silly changes to the source to avoid stupid compiler warni were reported on the Macintosh. The changes were from

while ((c = *(++ptr)) != 0 && c != '\n');
to
while ((c = *(++ptr)) != 0 && c != '\n');

Totally extraordinary, but if that's what it takes...

5. PCRE is being used in one environment where neither memmove() nor available. Added HAVE_BCOPY and an autoconf test for it; if neither HAVE_MEMMOVE nor HAVE_BCOPY is set, use a built-in emulation functio assumes the way PCRE uses memmove() (always moving upwards).

6. PCRE is being used in one environment where strchr() is not avail

was only one use in pcre.c, and writing it out to avoid strchr() pro faster code anyway.

Version 3.1 09-Feb-00

The only change in this release is the fixing of some bugs in Makefi the "install" target:

(1) It was failing to install pcreposix.h.

(2) It was overwriting the pcre.3 man page with the pcreposix.3 man

Version 3.0 01-Feb-00

1. Add support for the /+ modifier to perltest (to output \$` like it pcretest).

2. Add support for the /g modifier to perltest.

3. Fix pcretest so that it behaves even more like Perl for /g when t matches null strings.

4. Fix perltest so that it doesn't do unwanted things when fed an empattern. Perl treats empty patterns specially - it reuses the most r pattern, which is not what we want. Replace // by /(?#)/ in order to effect.

5. The POSIX interface was broken in that it was just handing over t captured string vector to pcre_exec(), but (since release 2.00) PCRE required a bigger vector, with some working space on the end. This m the POSIX wrapper now has to get and free some memory, and copy the

6. Added some simple autoconf support, placing the test data and the documentation in separate directories, re-organizing some of the information files, and making it build pcre-config (a GNU standard). libtool support for building PCRE as a shared library, which is now default.

7. Got rid of the leading zero in the definition of PCRE_MINOR becau 09 are not valid octal constants. Single digits will be used for min less than 10.

8. Defined REG_EXTENDED and REG_NOSUB as zero in the POSIX header, s existing programs that set these in the POSIX interface can use PCRE modification.

9. Added a new function, pcre_fullinfo() with an extensible interfac return all that pcre_info() returns, plus additional data. The pcre_function is retained for compatibility, but is considered to be obsc

10. Added experimental recursion feature (?R) to handle one common c Perl 5.6 will be able to do with $(?p\{\ldots\})$.

11. Added support for POSIX character classes like [:alpha:], which adopting.

Version 2.08 31-Aug-99

1. When startoffset was not zero and the pattern began with ".*", PC trying to match at the startoffset position, but instead was moving the next newline as if a previous match had failed.

2. pcretest was not making use of PCRE_NOTEMPTY when repeating for / and could get into a loop if a null string was matched other than at of the subject.

3. Added definitions of PCRE_MAJOR and PCRE_MINOR to pcre.h so the v be distinguished at compile time, and for completeness also added PC

5. Added Paul Sokolovsky's minor changes to make it easy to compile in GnuWin32 environments.

Version 2.07 29-Jul-99

1. The documentation is now supplied in plain text form and HTML as the form of man page sources.

2. C++ compilers don't like assigning (void *) values to other point In particular this affects malloc(). Although there is no problem in C, I've put in casts to keep C++ compilers happy.

3. Typo on pcretest.c; a cast of (unsigned char *) in the POSIX rege should be (const char *).

4. If NOPOSIX is defined, pcretest.c compiles without POSIX support. be useful for non-Unix systems who don't want to bother with the POS However, I haven't made this a standard facility. The documentation mention it, and the Makefile doesn't support it.

5. The Makefile now contains an "install" target, with editable dest the top of the file. The pcretest program is not installed. 6. pgrep -V now gives the PCRE version number and date.

7. Fixed bug: a zero repetition after a literal string (e.g. /abcde{ causing the entire string to be ignored, instead of just the last ch

8. If a pattern like /"([^\\"]+|\\.)*"/ is applied in the normal way non-matching string, it can take a very, very long time, even for st quite modest length, because of the nested recursion. PCRE now does some of these cases. It does this by remembering the last required 1 character in the pattern, and pre-searching the subject to ensure it before running the real match. In other words, it applies a heuristi some types of certain failure quickly, and in the above example, if with a string that has no trailing " it gives "no match" very quickl

9. A new runtime option PCRE_NOTEMPTY causes null string matches to other alternatives are tried instead.

Version 2.06 09-Jun-99

1. Change pcretest's output for amount of store used to show just th space, because the remainder (the data block) varies in size between 64-bit systems.

2. Added an extra argument to pcre_exec() to supply an offset in the start matching at. This allows lookbehinds to work when searching fo occurrences in a string.

3. Added additional options to pcretest for testing multiple occurre

- /+ outputs the rest of the string that follows a match
- /g loops for multiple occurrences, using the new startoffset ar
- /G loops for multiple occurrences by passing an incremented poi

4. PCRE wasn't doing the "first character" optimization for patterns with b or B, though it was doing it for other lookbehind assertion it wasn't noticing that a match for a pattern such as /\bxyz/ has to the letter 'x'. On long subject strings, this gives a significant sp

Version 2.05 21-Apr-99

1. Changed the type of magic_number from int to long int so that it properly on 16-bit systems.

2. Fixed a bug which caused patterns starting with .* not to work co when the subject string contained newline characters. PCRE was assum anchoring for such patterns in all cases, which is not correct becau not pass a newline unless PCRE_DOTALL is set. It now assumes anchori DOTALL is set at top level; otherwise it knows that patterns startin must be retried after every newline in the subject.

Version 2.04 18-Feb-99

1. For parenthesized subpatterns with repeats whose minimum was zero computation of the store needed to hold the pattern was incorrect (t If such patterns were nested a few deep, this could multiply and bec problem.

2. Added /M option to pcretest to show the memory requirement of a s pattern. Made -m a synonym of -s (which does this globally) for comp

3. Subpatterns of the form $(regex)\{n,m\}$ (i.e. limited maximum) were compiled in such a way that the backtracking after subsequent failur pessimal. Something like (a){0,3} was compiled as (a)?(a)?(a)? inste ((a)((a)(a)?)?)? with disastrous performance if the maximum was of a

Version 2.03 02-Feb-99

1. Fixed typo and small mistake in man page.

2. Added 4th condition (GPL supersedes if conflict) and created sepa LICENCE file containing the conditions.

3. Updated pcretest so that patterns such as /abc\/def/ work like th Perl, that is the internal $\$ allows the delimiter to be included in pattern. Locked out the use of $\$ as a delimiter. If $\$ immediately fo the final delimiter, add $\$ to the end of the pattern (to test the er

4. Added the convenience functions for extracting substrings after a match. Updated pcretest to make it able to test these functions.

Version 2.02 14-Jan-99

1. Initialized the working variables associated with each extraction their saving and restoring doesn't refer to uninitialized store.

2. Put dummy code into study.c in order to trick the optimizer of th compiler for OS/2 into generating correct code. Apparently IBM isn't fix the problem.

3. Pcretest: the timing code wasn't using LOOPREPEAT for timing exec

calls, and wasn't printing the correct value for compiling calls. In default value of LOOPREPEAT, and the number of significant figures i times.

4. Changed "/bin/rm" in the Makefile to "-rm" so it works on Windows

5. Renamed "deftables" as "dftables" to get it down to 8 characters, a building problem on Windows NT with a FAT file system.

Version 2.01 21-Oct-98

1. Changed the API for pcre_compile() to allow for the provision of to character tables built by pcre_maketables() in the current locale is passed, the default tables are used.

Version 2.00 24-Sep-98

1. Since the (>?) facility is in Perl 5.005, don't require PCRE_EXTR it any more.

2. Allow quantification of (?>) groups, and make it work correctly.

3. The first character computation wasn't working for (?>) groups.

4. Correct the implementation of Z (it is permitted to match on the end of the subject) and add 5.005's z, which really does match only very end of the subject.

5. Remove the \X "cut" facility; Perl doesn't have it, and (?> is ne

6. Remove the ability to specify CASELESS, MULTILINE, DOTALL, and DOLLAR_END_ONLY at runtime, to make it possible to implement the Per localized options. All options to pcre_study() were also removed.

7. Add other new features from 5.005:

\$(?<=	positive lookbehind
\$(? </td <td>negative lookbehind</td>	negative lookbehind
(?imsx-imsx)	added the unsetting capability
	such a setting is global if at outer level; local
(?imsx-imsx:)	non-capturing groups with option setting
(?(cond)re re)	conditional pattern matching

A backreference to itself in a repeated group matches the previou captured string.

8. General tidying up of studying (both automatic and via "study") consequential on the addition of new assertions.

9. As in 5.005, unlimited repeated groups that could match an empty are no longer faulted at compile time. Instead, the loop is forcibly runtime if any iteration does actually match an empty substring.

10. Include the RunTest script in the distribution.

11. Added tests from the Perl 5.005_02 distribution. This showed up discrepancies, some of which were old and were also with respect to have now been fixed.

Version 1.09 28-Apr-98

1. A negated single character class followed by a quantifier with a value of one (e.g. $[^x]{1,6}$) was not compiled correctly. This cc program crashes, or just wrong answers. This did not apply to negate containing more than one character, or to minima other than one.

Version 1.08 27-Mar-98

1. Add PCRE_UNGREEDY to invert the greediness of quantifiers.

2. Add (?U) and (?X) to set PCRE_UNGREEDY and PCRE_EXTRA respectivel latter must appear before anything that relies on it in the pattern.

Version 1.07 16-Feb-98

1. A pattern such as $/((a)^*)^*/$ was not being diagnosed as in error (repeat of a potentially empty string).

Version 1.06 23-Jan-98

1. Added Markus Oberhumer's little patches for C++.

2. Literal strings longer than 255 characters were broken.

Version 1.05 23-Dec-97

1. Negated character classes containing more than one character were PCRE_CASELESS was set at run time.

Version 1.04 19-Dec-97

1. Corrected the man page, where some "const" qualifiers had been om

 Made debugging output print "{0,xxx}" instead of just "{,xxx}" to input syntax.

3. Fixed memory leak which occurred when a regex with back reference matched with an offsets vector that wasn't big enough. The temporary that is used in this case wasn't being freed if the match failed.

4. Tidied pcretest to ensure it frees memory that it gets.

5. Temporary memory was being obtained in the case where the passed vector was exactly big enough.

6. Corrected definition of offsetof() from change 5 below.

7. I had screwed up change 6 below and broken the rules for the use setjmp(). Now fixed.

Version 1.03 18-Dec-97

1. A erroneous regex with a missing opening parenthesis was correctl diagnosed, but PCRE attempted to access brastack[-1], which could ca on some systems.

2. Replaced offsetof(real_pcre, code) by offsetof(real_pcre, code[0] it was reported that one broken compiler failed on the former becaus also an independent variable.

3. The erroneous regex a[]b caused an array overrun reference.

4. A regex ending with a one-character negative class (e.g. /[^k]\$/) fail on data ending with that character. (It was going on too far, a the next character, typically a binary zero.) This was specific to t optimized code for single-character negative classes.

5. Added a contributed patch from the TIN world which does the follo

+ Add an undef for memmove, in case the the system defines a macro

+ Add a definition of offsetof(), in case there isn't one. (I don'

the reason behind this - offsetof() is part of the ANSI standard it does no harm).

- + Reduce the ifdef's in pcre.c using macro DPRINTF, thereby elimin most of the places where whitespace preceded '#'. I have given u allowed the remaining 2 cases to be at the margin.
- + Rename some variables in pcre to eliminate shadowing. This seems pedantic, but does no harm, of course.

6. Moved the call to setjmp() into its own function, to get rid of ${\tt w}$ from gcc -Wall, and avoided calling it at all unless PCRE_EXTRA is u

7. Constructs such as $d{8}$ were compiling into the equivalent of $d{8}\d{0,65527}$ instead of $d{8}\d*$ which didn't make much differen outcome, but in this particular case used more store than had been a which caused the bug to be discovered because it threw up an interna

8. The debugging code in both pcre and pcretest for outputting the c form of a regex was going wrong in the case of back references follo curly-bracketed repeats.

Version 1.02 12-Dec-97

1. Typos in pcre.3 and comments in the source fixed.

2. Applied a contributed patch to get rid of places where it used to 'const' from variables, and fixed some signed/unsigned and uninitial variable warnings.

3. Added the "runtest" target to Makefile.

4. Set default compiler flag to -O2 rather than just -O.

Version 1.01 19-Nov-97

1. PCRE was failing to diagnose unlimited repeat of empty string for like $/([ab]^*)^*/$, that is, for classes with more than one character i

2. Likewise, it wasn't diagnosing patterns with "once-only" subpatte as /((?>a*))*/ (a PCRE_EXTRA facility).

Version 1.00 18-Nov-97

1. Added compile-time macros to support systems such as SunOS4 which memmove() or strerror() but have other things that can be used inste

2. Arranged that "make clean" removes the executables.

Version 0.99 27-Oct-97

1. Fixed bug in code for optimizing classes with only one character. initializing a 32-byte map regardless, which could cause it to run o of the memory it had got.

2. Added, conditional on PCRE_EXTRA, the proposed (?>REGEX) construc

Version 0.98 22-Oct-97

1. Fixed bug in code for handling temporary memory usage when there back references than supplied space in the ovector. This could cause

Version 0.97 21-Oct-97

1. Added the \X "cut" facility, conditional on PCRE_EXTRA.

2. Optimized negated single characters not to use a bit map.

3. Brought error texts together as macro definitions; clarified some fixed one that was wrong - it said "range out of order" when it mean escape sequence".

4. Changed some char * arguments to const char *.

5. Added PCRE_NOTBOL and PCRE_NOTEOL (from POSIX).

6. Added the POSIX-style API wrapper in pcreposix.a and testing faci pcretest.

Version 0.96 16-Oct-97

1. Added a simple "pgrep" utility to the distribution.

2. Fixed an incompatibility with Perl: "{" is now treated as a norma unless it appears in one of the precise forms "{ddd}", "{ddd,}", or where "ddd" means "one or more decimal digits".

3. Fixed serious bug. If a pattern had a back reference, but the cal pcre_exec() didn't supply a large enough ovector to record the relat identifying subpattern, the match always failed. PCRE now remembers of the largest back reference, and gets some temporary memory in whi the offsets during matching if necessary, in order to ensure that backreferences always work.

4. Increased the compatibility with Perl in a number of ways:

- (a) . no longer matches \n by default; an option PCRE_DOTALL is pr to request this handling. The option can be set at compile or
- (b) \$ matches before a terminating newline by default; an option PCRE_DOLLAR_ENDONLY is provided to override this (but not in m mode). The option can be set at compile or exec time.
- (c) The handling of \ followed by a digit other than 0 is now supp the same as Perl's. If the decimal number it represents is les or there aren't that many previous left capturing parentheses, escape is read. Inside a character class, it's always an octal even if it is a single digit.
- (d) An escaped but undefined alphabetic character is taken as a li unless PCRE_EXTRA is set. Currently this just reserves the rem escapes.
- (e) {0} is now permitted. (The previous item is removed from the c pattern).

5. Changed all the names of code files so that the basic parts are n than 10 characters, and abolished the teeny "globals.c" file.

6. Changed the handling of character classes; they are now done with bit map always.

7. Added the -d and /D options to pcretest to make it possible to lo internals of compilation without having to recompile pcre.

Version 0.95 23-Sep-97

1. Fixed bug in pre-pass concerning escaped "normal" characters such x20 at the start of a run of normal characters. These were being tr real characters, instead of the source characters being re-checked.

Version 0.94 18-Sep-97

1. The functions are now thread-safe, with the caveat that the globa containing pointers to malloc() and free() or alternative functions same for all threads.

2. Get pcre_study() to generate a bitmap of initial characters for n anchored patterns when this is possible, and use it if passed to pcr

Version 0.93 15-Sep-97

1. /(b)|(:+)/ was computing an incorrect first character.

2. Add pcre_study() to the API and the passing of pcre_extra to pcre but not actually doing anything yet.

3. Treat "-" characters in classes that cannot be part of ranges as as Perl does (e.g. [-az] or [az-]).

4. Set the anchored flag if a branch starts with .* or .*? because t all possible positions.

5. Split up into different modules to avoid including unneeded funct compiled binary. However, compile and exec are still in one module. function is split off.

6. The character tables are now in a separate module whose source is by an auxiliary program - but can then be edited by hand if required now no calls to isalnum(), isspace(), isdigit(), isxdigit(), tolower toupper() in the code.

7. Turn the malloc/free funtions variables into pcre_malloc and pcre make them global. Abolish the function for setting them, as the call set them directly.

Version 0.92 11-Sep-97

1. A repeat with a fixed maximum and a minimum of 1 for an ordinary (e.g. $/a\{1,3\}/)$ was broken (I mis-optimized it).

2. Caseless matching was not working in character classes if the cha the pattern were in upper case.

3. Make ranges like [W-c] work in the same way as Perl for caseless

4. Make PCRE_ANCHORED public and accept as a compile option.

5. Add an options word to pcre_exec() and accept PCRE_ANCHORED and PCRE_CASELESS at run time. Add escapes A and I to pcretest to caus pass them.

6. Give an error if bad option bits passed at compile or run time.

7. Add PCRE_MULTILINE at compile and exec time, and (?m) as well. Ad pcretest to cause it to pass that flag.

8. Add pcre_info(), to get the number of identifying subpatterns, th options, and the first character, if set.

9. Recognize C+ or C{n,m} where $n \ge 1$ as providing a fixed starting

Version 0.91 10-Sep-97

1. PCRE was failing to diagnose unlimited repeats of subpatterns tha match the empty string as in $/(a^*)^*/$. It was looping and ultimately

2. PCRE was looping on encountering an indefinitely repeated back re a subpattern that had matched an empty string, e.g. $/(a|)\1*/$. It no Perl does - treats the match as successful.

* * * *

README file for PCRE (Perl-compatible regular expression library)

The latest release of PCRE is always available from

ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre-xxx.tar

There is a mailing list for discussion about the development of PCRE

pcre-dev@exim.org

Please read the NEWS file if you are upgrading from a previous relea The contents of this README file are:

The PCRE APIS Documentation for PCRE Contributions by users of PCRE Building PCRE on non-Unix systems Building PCRE on Unix-like systems Retrieving configuration information on Unix-like systems Shared libraries on Unix-like systems Cross-compiling on Unix-like systems Using HP's ANSI C++ compiler (aCC) Making new tarballs Testing PCRE Character tables File manifest

The PCRE APIs

PCRE is written in C, and it has its own API. The distribution also set of C++ wrapper functions (see the pcrecpp man page for details), of Google Inc.

In addition, there is a set of C wrapper functions that are based on regular expression API (see the pcreposix man page). These end up in library called libpcreposix. Note that this just provides a POSIX ca interface to PCRE; the regular expressions themselves still follow P and semantics. The POSIX API is restricted, and does not give full a all of PCRE's facilities.

The header file for the POSIX-style functions is called pcreposix.h. official POSIX name is regex.h, but I did not want to risk possible with existing files of that name by distributing it that way. To use an existing program that uses the POSIX API, pcreposix.h will have t renamed or pointed at by a link.

If you are using the POSIX interface to PCRE and there is already a

library installed on your system, as well as worrying about the rege file (as mentioned above), you must also take care when linking prog ensure that they link with PCRE's libpcreposix library. Otherwise th up the POSIX functions of the same name from the other library.

One way of avoiding this confusion is to compile PCRE with the addit -Dregcomp=PCREregcomp (and similarly for the other POSIX functions) compiler flags (CFLAGS if you are using "configure" -- see below). T effect of renaming the functions so that the names no longer clash. you have to do the same thing for your applications, or write them u new names.

Documentation for PCRE

If you install PCRE in the normal way on a Unix-like system, you wil with a set of man pages whose names all start with "pcre". The one t called "pcre" lists all the others. In addition to these man pages, documentation is supplied in two other forms:

- 1. There are files called doc/pcre.txt, doc/pcregrep.txt, and doc/pcretest.txt in the source distribution. The first of these concatenation of the text forms of all the section 3 man pages those that summarize individual functions. The other two are th forms of the section 1 man pages for the pcregrep and pcretest These text forms are provided for ease of scanning with text ed similar tools. They are installed in prefix>/share/doc/pcre, w
- 2. A set of files containing all the documentation in HTML form, h in various ways, and rooted in a file called index.html, is dis doc/html and installed in <prefix>/share/doc/pcre/html.

Contributions by users of PCRE

You can find contributions from PCRE users in the directory

ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/Contrib

There is a README file giving brief descriptions of what they are. S complete in themselves; others are pointers to URLs containing relev Some of this material is likely to be well out-of-date. Several of t contributions provided support for compiling PCRE on various flavour Windows (I myself do not use Windows). Nowadays there is more Window in the standard distribution, so these contibutions have been archiv Building PCRE on non-Unix systems

For a non-Unix system, please read the comments in the file NON-UNIX though if your system supports the use of "configure" and "make" you able to build PCRE in the same way as for Unix-like systems.

PCRE has been compiled on many different operating systems. It shoul straightforward to build PCRE on any system that has a Standard C co library, because it uses only Standard C functions.

Building PCRE on Unix-like systems

If you are using HP's ANSI C++ compiler (aCC), please see the specia in the section entitled "Using HP's ANSI C++ compiler (aCC)" below.

The following instructions assume the use of the widely used "config make install" process. There is also some experimental support for " the PCRE distribution, but it is incomplete and not documented. Howe are a "cmake" user, you might want to try it.

To build PCRE on a Unix-like system, first run the "configure" comma PCRE distribution directory, with your current directory set to the where you want the files to be created. This command is a standard G "autoconf" configuration script, for which generic instructions are the file INSTALL.

Most commonly, people build PCRE within its own distribution directo this case, on many systems, just running "./configure" is sufficient the usual methods of changing standard defaults are available. For e

CFLAGS='-02 -Wall' ./configure --prefix=/opt/local

specifies that the C compiler should be run with the flags '-O2 -Wal of the default, and that "make install" should install PCRE under /o instead of the default /usr/local.

If you want to build in a different directory, just run "configure" directory as current. For example, suppose you have unpacked the PCR into /source/pcre/pcre-xxx, but you want to build it in /build/pcre/

cd /build/pcre/pcre-xxx
/source/pcre/pcre-xxx/configure

PCRE is written in C and is normally compiled as a C library. Howeve possible to build it as a C++ library, though the provided building does not have any features to support this.

There are some optional features that can be included or omitted fro library. You can read more about them in the pcrebuild man page.

- . If you want to suppress the building of the C++ wrapper library, y --disable-cpp to the "configure" command. Otherwise, when "configu it will try to find a C++ compiler and C++ header files, and if it it will try to build the C++ wrapper.
- . If you want to make use of the support for UTF-8 character strings you must add --enable-utf8 to the "configure" command. Without it, for handling UTF-8 is not included in the library. (Even when incl still has to be enabled by an option at run time.)
- . If, in addition to support for UTF-8 character strings, you want t support for the \P, \p, and \X sequences that recognize Unicode ch properties, you must add --enable-unicode-properties to the "confi command. This adds about 30K to the size of the library (in the fc property table); only the basic two-letter properties such as Lu a supported.
- . You can build PCRE to recognize either CR or LF or the sequence CR of the preceding, or any of the Unicode newline sequences as indic end of a line. Whatever you specify at build time is the default; of PCRE can change the selection at run time. The default newline is a single LF character (the Unix standard). You can specify the newline indicator by adding --enable-newline-is-cr or --enable-new or --enable-newline-is-crlf or --enable-newline-is-anycrlf or --enable-newline-is-any to the "configure" command, respectively.

If you specify --enable-newline-is-cr or --enable-newline-is-crlf, the standard tests will fail, because the lines in the test files LF. Even if the files are edited to change the line endings, there to be some failures. With --enable-newline-is-anycrlf or --enable-newline-is-any, many tests should succeed, but there may failures.

. When called via the POSIX interface, PCRE uses malloc() to get add storage for processing capturing parentheses if there are more tha them in a pattern. You can increase this threshold by setting, for

--with-posix-malloc-threshold=20

on the "configure" command.

. PCRE has a counter that can be set to limit the amount of resource If the limit is exceeded during a match, the match fails. The defa million. You can change the default by setting, for example,

--with-match-limit=500000

on the "configure" command. This is just the default; individual c pcre_exec() can supply their own value. There is more discussion o pcreapi man page.

. There is a separate counter that limits the depth of recursive fun during a matching process. This also has a default of ten million, essentially "unlimited". You can change the default by setting, fo

--with-match-limit-recursion=500000

Recursive function calls use up the runtime stack; running out of cause programs to crash in strange ways. There is a discussion abo sizes in the pcrestack man page.

- . The default maximum compiled pattern size is around 64K. You can i this by adding --with-link-size=3 to the "configure" command. You increase it even more by setting --with-link-size=4, but this is u ever to be necessary. Increasing the internal link size will reduc performance.
- . You can build PCRE so that its internal match() function that is c pcre_exec() does not call itself recursively. Instead, it uses mem obtained from the heap via the special functions pcre_stack_malloc pcre_stack_free() to save data that would otherwise be saved on th build PCRE like this, use

--disable-stack-for-recursion

on the "configure" command. PCRE runs more slowly in this mode, bu necessary in environments with limited stack sizes. This applies o pcre_exec() function; it does not apply to pcre_dfa_exec(), which use deeply nested recursion. There is a discussion about stack siz pcrestack man page.

. For speed, PCRE uses four tables for manipulating and identifying whose code point values are less than 256. By default, it uses a s tables for ASCII encoding that is part of the distribution. If you

--enable-rebuild-chartables

a program called dftables is compiled and run in the default C loc you obey "make". It builds a source file called pcre_chartables.c. not specify this option, pcre_chartables.c is created as a copy of pcre_chartables.c.dist. See "Character tables" below for further i

. It is possible to compile PCRE for use on systems that use EBCDIC default character code (as opposed to ASCII) by specifying

--enable-ebcdic

This automatically implies --enable-rebuild-chartables (see above)

The "configure" script builds the following files for the basic C li

- . Makefile is the makefile that builds the library
- . config.h contains build-time configuration options for the library
- . pcre.h is the public PCRE header file
- . pcre-config is a script that shows the settings of "configure" opt
- . libpcre.pc is data for the pkg-config command
- . libtool is a script that builds shared and/or static libraries
- . RunTest is a script for running tests on the basic C library
- . RunGrepTest is a script for running tests on the pcregrep command

Versions of config.h and pcre.h are distributed in the PCRE tarballs the names config.h.generic and pcre.h.generic. These are provided fo benefit of those who have to built PCRE without the benefit of "conf you use "configure", the .generic versions are not used.

If a C++ compiler is found, the following files are also built:

- . libpcrecpp.pc is data for the pkg-config command
- . pcrecpparg.h is a header file for programs that call PCRE via the
- . pcre_stringpiece.h is the header for the C++ "stringpiece" functio

The "configure" script also creates config.status, which is an execu script that can be run to recreate the configuration, and config.log contains compiler output from tests that "configure" runs.

Once "configure" has run, you can run "make". It builds two librarie libpcre and libpcreposix, a test program called pcretest, a demonstr program called pcredemo, and the pcregrep command. If a C++ compiler on your system, "make" also builds the C++ wrapper library, which is libpcrecpp, and some test programs called pcrecpp_unittest, pcre_scanner_unittest, and pcre_stringpiece_unittest. Building the C can be disabled by adding --disable-cpp to the "configure" command.

The command "make check" runs all the appropriate tests. Details of tests are given below in a separate section of this document.

You can use "make install" to install PCRE into live directories on system. The following are installed (file names are all relative to <prefix> that is set when "configure" is run):

Commands (bin): pcretest pcregrep pcre-config Libraries (lib): libpcre

```
libpcreposix
    libpcrecpp (if C++ support is enabled)
  Configuration information (lib/pkgconfig):
    libpcre.pc
    libpcrecpp.pc (if C++ support is enabled)
  Header files (include):
    pcre.h
    pcreposix.h
    pcre_scanner.h
    pcre_stringpiece.h ) if C++ support is enabled
    pcrecpp.h
                         )
    pcrecpparg.h
                         )
  Man pages (share/man/man{1,3}):
    pcregrep.1
    pcretest.1
    pcre.3
    pcre*.3 (lots more pages, all starting "pcre")
  HTML documentation (share/doc/pcre/html):
    index.html
    *.html (lots more pages, hyperlinked from index.html)
  Text file documentation (share/doc/pcre):
    AUTHORS
    COPYING
    ChangeLog
    LICENCE
    NEWS
    README
    pcre.txt (a concatenation of the man(3) pages)
pcretest.txt the pcretest man page
    pcregrep.txt the pcregrep man page
Note that the pcredemo program that is built by "configure" is *not*
anywhere. It is a demonstration for programmers wanting to use PCRE.
If you want to remove PCRE from your system, you can run "make unins
This removes all the files that "make install" installed. However, i
remove any directories, because these are often shared with other pr
```

Retrieving configuration information on Unix-like systems

Running "make install" installs the command pcre-config, which can b recall information about the PCRE configuration and installation. Fo

pcre-config --version

prints the version number, and

pcre-config --libs

outputs information about where the library is installed. This comma included in makefiles for programs that use PCRE, saving the program having to remember too many details.

The pkg-config command is another system for saving and retrieving i about installed libraries. Instead of separate commands for each lib single command is used. For example:

pkg-config --cflags pcre

The data is held in *.pc files that are installed in a directory cal <prefix>/lib/pkgconfig.

Shared libraries on Unix-like systems

The default distribution builds PCRE as shared libraries and static as long as the operating system supports shared libraries. Shared li support relies on the "libtool" script which is built as part of the "configure" process.

The libtool script is used to compile and link both shared and stati libraries. They are placed in a subdirectory called .libs when they built. The programs pcretest and pcregrep are built to use these uni libraries (by means of wrapper scripts in the case of shared librari you use "make install" to install shared libraries, pcregrep and pcr automatically re-built to use the newly installed shared libraries b installed themselves. However, the versions left in the build direct use the uninstalled libraries.

To build PCRE using static libraries only you must use --disable-sha configuring it. For example:

./configure --prefix=/usr/gnu --disable-shared

Then run "make" in the usual way. Similarly, you can use --disable-s build only shared libraries.

Cross-compiling on Unix-like systems

You can specify CC and CFLAGS in the normal way to the "configure" c

order to cross-compile PCRE for some other host. However, you should specify --enable-rebuild-chartables, because if you do, the dftables file is compiled and run on the local host, in order to generate the character tables (the pcre_chartables.c file). This will probably no because dftables.c needs to be compiled with the local compiler, not compiler.

When --enable-rebuild-chartables is not specified, pcre_chartables.c by making a copy of pcre_chartables.c.dist, which is a default set o that assumes ASCII code. Cross-compiling with the default tables sho a problem.

If you need to modify the character tables when cross-compiling, you move pcre_chartables.c.dist out of the way, then compile dftables.c run it on the local host to make a new version of pcre_chartables.c. Then when you cross-compile PCRE this new version of the tables will

Using HP's ANSI C++ compiler (aCC)

Unless C++ support is disabled by specifying the "--disable-cpp" opt "configure" script, you must include the "-AA" option in the CXXFLAG environment variable in order for the C++ components to compile corr

Also, note that the aCC compiler on PA-RISC platforms may have a def needed libraries fail to get included when specifying the "-AA" comp option. If you experience unresolved symbols when linking the C++ pr use the workaround of specifying the following environment variable running the "configure" script:

CXXLDFLAGS="-lstd_v2 -lCsup_v2"

Making new tarballs

The command "make dist" creates three PCRE tarballs, in tar.gz, tar. zip formats. The command "make distcheck" does the same, but then do build of the new distribution to ensure that it works.

If you have modified any of the man page sources in the doc director should first run the PrepareRelease script before making a distribut script creates the .txt and HTML forms of the documentation from the

Testing PCRE

To test the basic PCRE library on a Unix system, run the RunTest scr

created by the configuring process. There is also a script called Ru that tests the options of the pcregrep command. If the C++ wrapper l built, three test programs called pcrecpp_unittest, pcre_scanner_uni pcre_stringpiece_unittest are also built.

Both the scripts and all the program tests are run if you obey "make "make test". For other systems, see the instructions in NON-UNIX-USE

The RunTest script runs the pcretest test program (which is document own man page) on each of the testinput files in the testdata directo turn, and compares the output with the contents of the corresponding files. A file called testtry is used to hold the main output from pc (testsavedregex is also used as a working file). To run pcretest on the test files, give its number as an argument to RunTest, for examp

RunTest 2

The first test file can also be fed directly into the perltest.pl sc check that Perl gives the same results. The only difference you shou in the first few lines, where the Perl version is given instead of t version.

The second set of tests check pcre_fullinfo(), pcre_info(), pcre_stu pcre_copy_substring(), pcre_get_substring(), pcre_get_substring_list detection, and run-time flags that are specific to PCRE, as well as wrapper API. It also uses the debugging flags to check some of the i pcre_compile().

If you build PCRE with a locale setting that is not the standard C l character tables may be different (see next paragraph). In some case cause failures in the second set of tests. For example, in a locale isprint() function yields TRUE for characters in the range 128-255, [:isascii:] inside a character class defines a different set of char this shows up in this test as a difference in the compiled code, whi listed for checking. Where the comparison test output contains [\x00 test will contain [\x00-\xff], and similarly in some other cases. Th bug in PCRE.

The third set of tests checks pcre_maketables(), the facility for bu set of character tables for a specific locale and using them instead default tables. The tests make use of the "fr_FR" (French) locale. E running the test, the script checks for the presence of this locale the "locale" command. If that command fails, or if it doesn't includ in the list of available locales, the third test cannot be run, and is output to say why. If running this test produces instances of the

** Failed to set locale "fr_FR"

in the comparison output, it means that locale is not available on y despite being listed by "locale". This does not mean that PCRE is br

[If you are trying to run this test on Windows, you may be able to g work by changing "fr_FR" to "french" everywhere it occurs.]

The fourth test checks the UTF-8 support. It is not run automaticall PCRE is built with UTF-8 support. To do this you must set --enable-u running "configure". This file can be also fed directly to the perlt provided you are running Perl 5.8 or higher. (For Perl 5.6, a small commented in the script, can be be used.)

The fifth test checks error handling with UTF-8 encoding, and intern features of PCRE that are not relevant to Perl.

The sixth test checks the support for Unicode character properties. run automatically unless PCRE is built with Unicode property support this you must set --enable-unicode-properties when running "configur

The seventh, eighth, and ninth tests check the pcre_dfa_exec() alter matching function, in non-UTF-8 mode, UTF-8 mode, and UTF-8 mode wit property support, respectively. The eighth and ninth tests are not r automatically unless PCRE is build with the relevant support.

Character tables

For speed, PCRE uses four tables for manipulating and identifying ch whose code point values are less than 256. The final argument of the pcre_compile() function is a pointer to a block of memory containing concatenated tables. A call to pcre_maketables() can be used to gene of tables in the current locale. If the final argument for pcre_comp passed as NULL, a set of default tables that is built into the binar

The source file called pcre_chartables.c contains the default set of default, this is created as a copy of pcre_chartables.c.dist, which tables for ASCII coding. However, if --enable-rebuild-chartables is for ./configure, a different version of pcre_chartables.c is built b program dftables (compiled from dftables.c), which uses the ANSI C c handling functions such as isalnum(), isalpha(), isupper(), islower(build the table sources. This means that the default C locale which your system will control the contents of these default tables. You c the default tables by editing pcre_chartables.c and then re-building you do this, you should take care to ensure that the file does not g automatically re-generated. The best way to do this is to move pcre_chartables.c.dist out of the way and replace it with your custo tables.

When the dftables program is run as a result of --enable-rebuild-cha it uses the default C locale that is set on your system. It does not attention to the LC_xxx environment variables. In other words, it us system's default locale rather than whatever the compiling user happ set. If you really do want to build a source set of character tables locale that is specified by the LC_xxx variables, you can run the df program by hand with the -L option. For example:

./dftables -L pcre_chartables.c.special

The first two 256-byte tables provide lower casing and case flipping respectively. The next table consists of three 32-byte bit maps whic digits, "word" characters, and white space, respectively. These are building 32-byte bit maps that represent character classes for code than 256.

The final 256-byte table has bits indicating various character types follows:

- 1 white space character
- 2 letter
- 4 decimal digit
- 8 hexadecimal digit
- 16 alphanumeric or '_'
- 128 regular expression metacharacter or binary zero

You should not alter the set of characters that contain the 128 bit, will cause PCRE to malfunction.

File manifest

The distribution should contain the following files:

(A) Source files of the PCRE library functions and their headers:

dftables.c	auxiliary program for building pcre_charta whenenable-rebuild-chartables is spec
pcre_chartables.c.dist	a default set of character tables that ass coding; used, unlessenable-rebuild-ch specified, by copying to pcre_chartables
<pre>pcreposix.c pcre_compile.c pcre_config.c pcre_dfa_exec.c pcre_exec.c pcre_fullinfo.c pcre_get.c pcre_globals.c pcre_info.c</pre>))))) sources for the functions in the library) and some internal functions that they)

<pre>pcre_maketables.c pcre_newline.c pcre_ord2utf8.c pcre_refcount.c pcre_study.c pcre_tables.c pcre_try_flipped.c pcre_ucp_searchfuncs.c pcre_valid_utf8.c pcre_version.c pcre_version.c pcre_printint.src pcre_printint.src pcre_internal.h ucp.h ucpinternal.h ucptable.h</pre>	<pre>))))))))))))))))))))))))))</pre>	
config.h.in	template for config.h, which is built by "	
pcrecpp.h pcrecpparg.h.in pcre_scanner.h pcrecpp.cc pcre_scanner.cc	public header file for the C++ wrapper template for another C++ header file public header file for C++ scanner functic)) source for the C++ wrapper library	
<pre>pcre_stringpiece.h.in pcre_stringpiece.cc</pre>	template for pcre_stringpiece.h, the heade C++ stringpiece functions source for the C++ stringpiece functions	
(B) Source files for programs that use PCRE:		
pcredemo.c pcregrep.c pcretest.c	simple demonstration of coding calls to PC source of a grep utility that uses PCRE comprehensive test program	
(C) Auxiliary files:		
132html AUTHORS ChangeLog CleanTxt Detrail HACKING INSTALL LICENCE COPYING Makefile.in	<pre>script to turn "man" pages into HTML information about the author of PCRE log of changes to the code script to clean nroff output for txt man p script to remove trailing spaces some notes about the internals of PCRE generic installation instructions conditions for the use of PCRE the same, using GNU's standard name) template for Unix Makefile, which is bui</pre>	

"configure" Makefile.am) the automake input that was used to crea Makefile.in) NEWS important changes in this release notes on building PCRE on non-Unix systems NON-UNIX-USE PrepareRelease script to make preparations for "make dist this file README RunTest a Unix shell script for running tests a Unix shell script for pcregrep tests RunGrepTest aclocal.m4 m4 macros (generated by "aclocal") config.guess) files used by libtool, used only when building a shared libra config.sub) configure a configuring shell script (built by autoc) the autoconf input that was used to buil configure.ac "configure" and config.h) script to find program dependencies, gen depcomp automake) man page sources for the PCRE functions doc/*.3doc/*.1 man page sources for pcregrep and pcretest doc/index.html.src the base HTML page doc/html/* HTML documentation plain text version of the man pages doc/pcre.txt plain text documentation of test program doc/pcretest.txt doc/perltest.txt plain text documentation of Perl test prog install-sh a shell script for installing files libpcre.pc.in template for libpcre.pc for pkg-config template for libpcrecpp.pc for pkg-config libpcrecpp.pc.in ltmain.sh file used to build a libtool script missing) common stub for a few missing GNU progra installing, generated by automake) script for making install directories mkinstalldirs perltest.pl Perl test program pcre-config.in source of script which retains PCRE inform pcrecpp_unittest.cc) test programs for the C++ wrapper pcre scanner unittest.cc pcre_stringpiece_unittest.cc) testdata/testinput* test data for main library tests testdata/testoutput* expected test results testdata/grep* input and output for pcregrep tests (D) Auxiliary files for cmake support CMakeLists.txt config-cmake.h.in (E) Auxiliary files for VPASCAL makevp.bat makevp_c.txt makevp_l.txt

pcregexp.pas

(F) Auxiliary files for building PCRE "by hand"

pcre.h.generic) a version of the public PCRE header file
) for use in non-"configure" environment
config.h.generic) a version of config.h for use in non-"co
) environments

(F) Miscellaneous

RunTest.bat a script for running tests under Windows

Philip Hazel Email local part: ph10 Email domain: cam.ac.uk Last updated: 24 April 2007 Installation Instructions

Copyright (C) 1994, 1995, 1996, 1999, 2000, 2001, 2002, 2004, 2005, 2006 Free Software Foundation, Inc.

This file is free documentation; the Free Software Foundation gives unlimited permission to copy, distribute and modify it.

Basic Installation

Briefly, the shell commands `./configure; make; make install' should configure, build, and install this package. The following more-detailed instructions are generic; see the `README' file for instructions specific to this package.

The `configure' shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a `Makefile' in each directory of the package It may also create one or more `.h' files containing system-dependen definitions. Finally, it creates a shell script `config.status' tha you can run in the future to recreate the current configuration, and file `config.log' containing compiler output (useful mainly for debugging `configure').

It can also use an optional file (typically called `config.cache' and enabled with `--cache-file=config.cache' or simply `-C') that sa the results of its tests to speed up reconfiguring. Caching is disabled by default to prevent problems with accidental use of stale cache files.

If you need to do unusual things to compile the package, please t to figure out how `configure' could check whether to do them, and ma diffs or instructions to the address given in the `README' so they c be considered for the next release. If you are using the cache, and some point `config.cache' contains results you don't want to keep, y may remove or edit it.

The file `configure.ac' (or `configure.in') is used to create `configure' by a program called `autoconf'. You need `configure.ac' you want to change it or regenerate `configure' using a newer versic of `autoconf'.

The simplest way to compile this package is:

 `cd' to the directory containing the package's source code and `./configure' to configure the package for your system.

Running `configure' might take a while. While running, it prin

some messages telling which features it is checking for.

- 2. Type `make' to compile the package.
- 3. Optionally, type `make check' to run any self-tests that come w the package.
- 4. Type `make install' to install the programs and any data files documentation.
- 5. You can remove the program binaries and object files from the source code directory by typing `make clean'. To also remove t files that `configure' created (so you can compile the package a different kind of computer), type `make distclean'. There is also a `make maintainer-clean' target, but that is intended mai for the package's developers. If you use it, you may have to g all sorts of other programs in order to regenerate files that c with the distribution.

Compilers and Options

Some systems require unusual options for compilation or linking that `configure' script does not know about. Run `./configure --help' fo details on some of the pertinent environment variables.

You can give `configure' initial values for configuration paramet by setting variables in the command line or in the environment. Her is an example:

./configure CC=c99 CFLAGS=-g LIBS=-lposix

*Note Defining Variables::, for more details.

Compiling For Multiple Architectures

You can compile the package for more than one kind of computer at th same time, by placing the object files for each architecture in thei own directory. To do this, you can use GNU `make'. `cd' to the directory where you want the object files and executables to go and the `configure' script. `configure' automatically checks for the source code in the directory that `configure' is in and in `..'.

With a non-GNU `make', it is safer to compile the package for one architecture at a time in the source code directory. After you have installed the package for one architecture, use `make distclean' bef reconfiguring for another architecture.

Installation Names

By default, `make install' installs the package's commands under `/usr/local/bin', include files under `/usr/local/include', etc. Yo can specify an installation prefix other than `/usr/local' by giving `configure' the option `--prefix=PREFIX'.

You can specify separate installation prefixes for architecture-specific files and architecture-independent files. If pass the option `--exec-prefix=PREFIX' to `configure', the package u PREFIX as the prefix for installing programs and libraries. Documentation and other data files still use the regular prefix.

In addition, if you use an unusual directory layout you can give options like `--bindir=DIR' to specify different values for particul kinds of files. Run `configure --help' for a list of the directorie you can set and what kinds of files go in them.

If the package supports it, you can cause programs to be installe with an extra prefix or suffix on their names by giving `configure' option `--program-prefix=PREFIX' or `--program-suffix=SUFFIX'.

Optional Features

Some packages pay attention to `--enable-FEATURE' options to `configure', where FEATURE indicates an optional part of the package They may also pay attention to `--with-PACKAGE' options, where PACKA is something like `gnu-as' or `x' (for the X Window System). The `README' should mention any `--enable-' and `--with-' options that t package recognizes.

For packages that use the X Window System, `configure' can usuall find the X include and library files automatically, but if it doesn' you can use the `configure' options `--x-includes=DIR' and `--x-libraries=DIR' to specify their locations.

Specifying the System Type

There may be some features `configure' cannot figure out automatical but needs to determine by the type of machine the package will run o Usually, assuming the package is built to be run on the _same_ architectures, `configure' can figure that out, but if it prints a message saying it cannot guess the machine type, give it the `--build=TYPE' option. TYPE can either be a short name for the syst type, such as `sun4', or a canonical name which has the form:

CPU-COMPANY-SYSTEM

where SYSTEM can have one of these forms:

OS KERNEL-OS

See the file `config.sub' for the possible values of each field. `config.sub' isn't included in this package, then this package doesn need to know the machine type.

If you are _building_ compiler tools for cross-compiling, you sho use the option `--target=TYPE' to select the type of system they wil produce code for.

If you want to _use_ a cross compiler, that generates code for a platform different from the build platform, you should specify the "host" platform (i.e., that on which the generated programs will eventually be run) with `--host=TYPE'.

Sharing Defaults

If you want to set default values for `configure' scripts to share, can create a site shell script called `config.site' that gives defau values for variables like `CC', `cache_file', and `prefix'. `configure' looks for `PREFIX/share/config.site' if it exists, then `PREFIX/etc/config.site' if it exists. Or, you can set the `CONFIG_SITE' environment variable to the location of the site scrip A warning: not all `configure' scripts look for a site script.

Defining Variables

Variables not defined in a site shell script can be set in the environment passed to `configure'. However, some packages may run configure again during the build, and the customized values of these variables may be lost. In order to avoid this problem, you should s them in the `configure' command line, using `VAR=value'. For exampl

./configure CC=/usr/local2/bin/gcc

causes the specified `gcc' to be used as the C compiler (unless it i overridden in the site shell script).

Unfortunately, this technique does not work for `CONFIG_SHELL' due t an Autoconf bug. Until the bug is fixed you can use this workaround

CONFIG_SHELL=/bin/bash /bin/bash ./configure CONFIG_SHELL=/bin/

`configure' recognizes the following options to control how it opera `--help' `-h' Print a summary of the options to `configure', and exit. `--version' `-V' Print the version of Autoconf used to generate the `configure' script, and exit. `--cache-file=FILE' Enable the cache: use and save the results of the tests in FILE traditionally `config.cache'. FILE defaults to `/dev/null' to disable caching. `--config-cache' `-C' Alias for `--cache-file=config.cache'. `--quiet' `--silent' `-q' Do not print messages saying which checks are being made. To suppress all normal output, redirect it to `/dev/null' (any err messages will still be shown). `--srcdir=DIR' Look for the package's source code in directory DIR. Usually `configure' can determine that directory automatically. `configure' also accepts some other, not widely useful, options. Ru `configure --help' for more details.

Compiling PCRE on non-Unix systems

This document contains the following sections:

General Generic instructions for the PCRE C library The C++ wrapper functions Building for virtual Pascal Comments about Win32 builds Building under Windows with BCC5.5 Building PCRE on OpenVMS

GENERAL

I (Philip Hazel) have no knowledge of Windows or VMS sytems and how libraries work. The items in the PCRE distribution and Makefile that anything other than Unix-like systems are untested by me.

There are some other comments and files in the Contrib directory on site that you may find useful. See

ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/Contrib

If you want to compile PCRE for a non-Unix system (especially for a does not support "configure" and "make" files), note that the basic library consists entirely of code written in Standard C, and so shou successfully on any system that has a Standard C compiler and librar wrapper functions are a separate issue (see below).

The PCRE distribution contains some experimental support for "cmake" is incomplete and not documented. However if you are a "cmake" user like to try building with "cmake".

GENERIC INSTRUCTIONS FOR THE PCRE C LIBRARY

The following are generic comments about building the PCRE C library

(1) Copy or rename the file config.h.generic as config.h, and edit t settings that it contains to whatever is appropriate for your en In particular, if you want to force a specific value for newline define the NEWLINE macro.

An alternative approach is not to edit config.h, but to use -D c compiler command line to make any changes that you need.

NOTE: There have been occasions when the way in which certain pa

config.h are used has changed between releases. (In the configur world, this is handled automatically.) When upgrading to a new r are strongly advised to review config.h.generic before re-using had previously.

- (2) Copy or rename the file pcre.h.generic as pcre.h.
- (3) EITHER:

Copy or rename file pcre_chartables.c.dist as pcre_chartables.

OR:

Compile dftables.c as a stand-alone program, and then run it w single argument "pcre_chartables.c". This generates a set of s character tables and writes them to that file. The tables are using the default C locale for your system. If you want to use that is specified by LC_xxx environment variables, add the -L the dftables command. You must use this method if you are buil a system that uses EBCDIC code.

The tables in pcre_chartables.c are defaults. The caller of PCRE specify alternative tables at run time.

(4) Compile the following source files:

pcre chartables.c pcre_compile.c pcre config.c pcre dfa exec.c pcre exec.c pcre_fullinfo.c pcre get.c pcre_globals.c pcre info.c pcre_maketables.c pcre_newline.c pcre_ord2utf8.c pcre_refcount.c pcre_study.c pcre tables.c pcre_try_flipped.c pcre_ucp_searchfuncs.c pcre valid utf8.c pcre_version.c pcre_xclass.c

Now link them all together into an object library in whichever f system keeps such libraries. This is the basic PCRE C library. I system has static and shared libraries, you may have to do this each type.

- (5) Similarly, compile pcreposix.c and link it (on its own) as the p library.
- (6) Compile the test program pcretest.c. This needs the functions in pcre and pcreposix libraries when linking.
- (7) Run pcretest on the testinput files in the testdata directory, a that the output matches the corresponding testoutput files. Note supplied files are in Unix format, with just LF characters as li terminators. You may need to edit them to change this if your sy different convention.
- (8) If you want to use the pcregrep command, compile and link pcregr uses only the basic PCRE library (it does not need the pcreposix

THE C++ WRAPPER FUNCTIONS

The PCRE distribution also contains some C++ wrapper functions and t contributed by Google Inc. On a system that can use "configure" and the functions are automatically built into a library called pcrecpp. be straightforward to compile the .cc files manually on other system files called xxx_unittest.cc are test programs for each of the corre xxx.cc files.

BUILDING FOR VIRTUAL PASCAL

A script for building PCRE using Borland's C++ compiler for use with was contributed by Alexander Tokarev. Stefan Weber updated the scrip additional files. The following files in the distribution are for bu for use with VP/Borland: makevp_c.txt, makevp_l.txt, makevp.bat, pcr

COMMENTS ABOUT WIN32 BUILDS

There are two ways of building PCRE using the "configure, make, make paradigm on Windows systems: using MinGW or using Cygwin. These are the same thing; they are completely different from each other. There some experimental, undocumented support for building using "cmake", might like to try if you are familiar with "cmake". However, at the time, the "cmake" process builds only a static library (not a dll), tests are not automatically run.

The MinGW home page (http://www.mingw.org/) says this:

MinGW: A collection of freely available and freely distributable w specific header files and import libraries combined with GNU tools allow one to produce native Windows programs that do not rely on a 3rd-party C runtime DLLs. The Cygwin home page (http://www.cygwin.com/) says this:

Cygwin is a Linux-like environment for Windows. It consists of two

. A DLL (cygwin1.dll) which acts as a Linux API emulation layer pr substantial Linux API functionality

. A collection of tools which provide Linux look and feel.

The Cygwin DLL currently works with all recent, commercially relea bit and 64 bit versions of Windows, with the exception of Windows

On both MinGW and Cygwin, PCRE should build correctly using:

./configure && make && make install

This should create two libraries called libpcre and libpcreposix, an have enabled building the C++ wrapper, a third one called libpcrecpp independent libraries: when you like with libpcreposix or libpcrecpp also link with libpcre, which contains the basic functions. (Some ea releases of PCRE included the basic libpcre functions in libpcreposi longer happens.)

If you want to statically link your program against a non-dll .a fil define PCRE_STATIC before including pcre.h, otherwise the pcre_mallc pcre_free() exported functions will be declared __declspec(dllimport unwanted results.

Using Cygwin's compiler generates libraries and executables that dep cygwin1.dll. If a library that is generated this way is distributed, cygwin1.dll has to be distributed as well. Since cygwin1.dll is unde licence, this forces not only PCRE to be under the GPL, but also the application. A distributor who wants to keep their own code propriet purchase an appropriate Cygwin licence.

MinGW has no such restrictions. The MinGW compiler generates a libra executable that can run standalone on Windows without any third part licensing issues.

But there is more complication:

If a Cygwin user uses the -mno-cygwin Cygwin gcc flag, what that rea to tell Cygwin's gcc to use the MinGW gcc. Cygwin's gcc is only acti front end to MinGW's gcc (if you install Cygwin's gcc, you get both gcc and MinGW's gcc). So, a user can:

. Build native binaries by using MinGW or by getting Cygwin and usin -mno-cygwin.

. Build binaries that depend on cygwin1.dll by using Cygwin with the compiler flags.

The test files that are supplied with PCRE are in Unix format, with characters as line terminators. It may be necessary to change the li terminators in order to get some of the tests to work. We hope to im things in this area in future.

BUILDING UNDER WINDOWS WITH BCC5.5

Michael Roy sent these comments about building PCRE under Windows wi

Some of the core BCC libraries have a version of PCRE from 1998 bu which can lead to pcre_exec() giving an erroneous PCRE_ERROR_NULL version mismatch. I'm including an easy workaround below, if you'd include it in the non-unix instructions:

When linking a project with BCC5.5, pcre.lib must be included befo the libraries cw32.lib, cw32i.lib, cw32mt.lib, and cw32mti.lib on line.

BUILDING PCRE ON OPENVMS

Dan Mooney sent the following comments about building PCRE on OpenVM relate to an older version of PCRE that used fewer source files, so commands will need changing. See the current list of source files ab

"It was quite easy to compile and link the library. I don't have a f make file but the attached file [reproduced below] contains the Open commands I used to build the library. I had to add #define POSIX_MALLOC_THRESHOLD 10 to pcre.h since it was not defined anywher

The library was built on: O/S: HP OpenVMS v7.3-1 Compiler: Compaq C v6.5-001-48BCD Linker: vA13-01

The test results did not match 100% due to the issues you mention in documentation regarding isprint(), iscntrl(), isgraph() and ispunct(modified some of the character tables temporarily and was able to ge results to match. Tests using the fr locale did not match since I do that locale loaded. The study size was always reported to be 3 less value in the standard test output files."

\$! This DCL procedure builds PCRE on OpenVMS

\$!

\$! I followed the instructions in the non-unix-use file in the distr

```
$!
$ COMPILE == "CC/LIST/NOMEMBER ALIGNMENT/PREFIX LIBRARY ENTRIES=ALL
$ COMPILE DFTABLES.C
$ LINK/EXE=DFTABLES.EXE DFTABLES.OBJ
$ RUN DFTABLES.EXE/OUTPUT=CHARTABLES.C
$ COMPILE MAKETABLES.C
$ COMPILE GET.C
$ COMPILE STUDY.C
$! I had to set POSIX_MALLOC_THRESHOLD to 10 in PCRE.H since the sym
$! did not seem to be defined anywhere.
$! I edited pcre.h and added #DEFINE SUPPORT_UTF8 to enable UTF8 sup
$ COMPILE PCRE.C
$ LIB/CREATE PCRE MAKETABLES.OBJ, GET.OBJ, STUDY.OBJ, PCRE.OBJ
$! I had to set POSIX_MALLOC_THRESHOLD to 10 in PCRE.H since the sym
$! did not seem to be defined anywhere.
$ COMPILE PCREPOSIX.C
$ LIB/CREATE PCREPOSIX PCREPOSIX.0BJ
$ COMPILE PCRETEST.C
$ LINK/EXE=PCRETEST.EXE PCRETEST.OBJ, PCRE/LIB, PCREPOSIX/LIB
$! C programs that want access to command line arguments must be
$! defined as a symbol
$ PCRETEST :== "$ SYS$ROADSUSERS:[DMOONEY.REGEXP]PCRETEST.EXE"
$! Arguments must be enclosed in quotes.
$ PCRETEST "-C"
$! Test results:
$!
     The test results did not match 100%. The functions isprint(), i
$!
$!
     isgraph() and ispunct() on OpenVMS must not produce the same re
$!
     as the system that built the test output files provided with th
$!
     distribution.
$!
$!
     The study size did not match and was always 3 less on OpenVMS.
$!
$!
     Locale could not be set to fr
$!
_____
Last Updated: 13 June 2007
```

Technical Notes about PCRE

These are very rough technical notes that record potentially useful about PCRE internals.

Historical note 1

Many years ago I implemented some regular expression functions to an suggested by Martin Richards. These were not Unix-like in form, and restricted in what they could do by comparison with Perl. The intere about the algorithm was that the amount of space required to hold th form of an expression was known in advance. The code to apply an exp not operate by backtracking, as the original Henry Spencer code and Perl code does, but instead checked all possibilities simultaneously a list of current states and checking all of them as it advanced thr subject string. In the terminology of Jeffrey Friedl's book, it was algorithm", though it was not a traditional Finite State Machine (FS the pattern was all used up, all remaining states were possible matc the one matching the longest subset of the subject string was chosen not necessarily maximize the individual wild portions of the pattern expected in Unix and Perl-style regular expressions.

Historical note 2

By contrast, the code originally written by Henry Spencer (which was subsequently heavily modified for Perl) compiles the expression twic a dummy mode in order to find out how much store will be needed, and real. (The Perl version probably doesn't do this any more; I'm talki the original library.) The execution function operates by backtracki maximizing (or, optionally, minimizing in Perl) the amount of the su matches individual wild portions of the pattern. This is an "NFA alg Friedl's terminology.

OK, here's the real stuff

For the set of functions that form the "basic" PCRE library (which a unrelated to those mentioned above), I tried at first to invent an a that used an amount of store bounded by a multiple of the number of in the pattern, to save on compiling time. However, because of the g complexity in Perl regular expressions, I couldn't do this. In any c first pass through the pattern is helpful for other reasons.

Computing the memory requirement: how it was

Up to and including release 6.7, PCRE worked by running a very degen

pass to calculate a maximum store size, and then a second pass to do compile - which might use a bit less than the predicted amount of me idea was that this would turn out faster than the Henry Spencer code the first pass is degenerate and the second pass can just store stuf into the vector, which it knows is big enough.

Computing the memory requirement: how it is

By the time I was working on a potential 6.8 release, the degenerate had become very complicated and hard to maintain. Indeed one of the things I did for 6.8 was to fix Yet Another Bug in the memory comput I had a flash of inspiration as to how I could run the real compile a "fake" mode that enables it to compute how much memory it would ne actually only ever using a few hundred bytes of working memory, and many tests of the mode that might slow it down. So I re-factored the functions to work this way. This got rid of about 600 lines of sourc should make future maintenance and development easier. As this was s change, I never released 6.8, instead upping the number to 7.0 (othe major changes are also present in the 7.0 release).

A side effect of this work is that the previous limit of 200 on the depth of parentheses was removed. However, there is a downside: pcre runs more slowly than before (30% or more, depending on the pattern) is doing a full analysis of the pattern. My hope is that this is not issue.

Traditional matching function

The "traditional", and original, matching function is called pcre_ex it implements an NFA algorithm, similar to the original Henry Spence and the way that Perl works. Not surprising, since it is intended to compatible with Perl as possible. This is the function most users of use most of the time.

Supplementary matching function

From PCRE 6.0, there is also a supplementary matching function calle pcre_dfa_exec(). This implements a DFA matching algorithm that searc simultaneously for all possible matches that start at one point in t string. (Going back to my roots: see Historical Note 1 above.) This intreprets the same compiled pattern data as pcre_exec(); however, n facilities are available, and those that are do not always work in q same way. See the user documentation for details.

The algorithm that is used for pcre_dfa_exec() is not a traditional because it may have a number of states active at one time. More work needed at compile time to produce a traditional FSM where only one s

ever active at once. I believe some other regex matchers work this w

Format of compiled patterns

The compiled form of a pattern is a vector of bytes, containing item variable length. The first byte in an item is an opcode, and the len item is either implicit in the opcode or contained in the data bytes follow it.

In many cases below "two-byte" data values are specified. This is in a default when the number is an offset within the compiled pattern. compiled to use 3-byte or 4-byte values for these offsets (impairing performance). This is necessary only when patterns whose compiled le greater than 64K are going to be processed. In this description, we "normal" compilation options. "Two-byte" data values that are counts quantifiers) are always just two bytes.

A list of all the opcodes follows:

Opcodes with no following data

These items are all just one byte long

OP_END	end of pattern
OP_ANY	match any character
OP_ANYBYTE	match any single byte, even in UTF-8 mode
OP_SOD	match start of data: \A
OP_SOM,	start of match (subject + offset): \G
OP_SET_SOM,	set start of match (\K)
OP_CIRC	<pre>^ (start of data, or after \n in multiline)</pre>
OP_NOT_WORD_BOUNDARY	\W
OP_WORD_BOUNDARY	\w
OP_NOT_DIGIT	\D
OP_DIGIT	\d
OP_NOT_HSPACE	\H
OP_HSPACE	\h
OP_NOT_WHITESPACE	\S
OP_WHITESPACE	\s
OP_NOT_VSPACE	\V
0P_VSPACE	\v
OP_NOT_WORDCHAR	\W
OP_WORDCHAR	\w
OP_EODN	match end of data or \n at end: \Z
OP_EOD	match end of data: \z
OP_DOLL	<pre>\$ (end of data, or before \n in multiline)</pre>
OP_EXTUNI	match an extended Unicode character
OP_ANYNL	match any Unicode newline sequence
	- •

Repeating single characters

The common repeats (*, +, ?) when applied to a single character use following opcodes:

OP_STAR OP_MINSTAR OP_POSSTAR OP_PLUS OP_MINPLUS OP_POSPLUS OP_QUERY OP_MINQUERY OP_POSQUERY

In ASCII mode, these are two-byte items; in UTF-8 mode, the length i Those with "MIN" in their name are the minimizing versions. Those wi their names are possessive versions. Each is followed by the charact to be repeated. Other repeats make use of

OP_UPTO OP_MINUPTO OP_POSUPTO OP_EXACT

which are followed by a two-byte count (most significant first) and repeated character. OP_UPTO matches from 0 to the given number. A re non-zero minimum and a fixed maximum is coded as an OP_EXACT followe OP_UPTO (or OP_MINUPTO or OPT_POSUPTO).

Repeating character types

Repeats of things like \d are done exactly as for single characters, that instead of a character, the opcode for the type is stored in th byte. The opcodes are:

OP_TYPESTAR OP_TYPEMINSTAR OP_TYPEPOSSTAR OP_TYPEPLUS OP_TYPEMINPLUS OP_TYPEPOSPLUS OP_TYPEQUERY OP_TYPEMINQUERY OP_TYPEPOSQUERY OP_TYPEUPTO OP_TYPEMINUPTO OP_TYPEPOSUPTO OP_TYPEEXACT

Match by Unicode property

OP_PROP and OP_NOTPROP are used for positive and negative matches of character by testing its Unicode property (the p and P escape sequ Each is followed by two bytes that encode the desired property as a value.

Repeats of these items use the OP_TYPESTAR etc. set of opcodes, foll three bytes: OP_PROP or OP_NOTPROP and then the desired property typ value.

Matching literal characters

The OP_CHAR opcode is followed by a single character that is to be m casefully. For caseless matching, OP_CHARNC is used. In UTF-8 mode, character may be more than one byte long. (Earlier versions of PCRE multi-character strings, but this was changed to allow some new feat added.)

Character classes

If there is only one character, OP_CHAR or OP_CHARNC is used for a p class, and OP_NOT for a negative one (that is, for something like [^ However, in UTF-8 mode, the use of OP_NOT applies only to characters values < 128, because OP_NOT is confined to single bytes.

Another set of repeating opcodes (OP_NOTSTAR etc.) are used for a re negated, single-character class. The normal ones (OP_STAR etc.) are repeated positive single-character class.

When there's more than one character in a class and all the characte than 256, OP_CLASS is used for a positive class, and OP_NCLASS for a one. In either case, the opcode is followed by a 32-byte bit map con bit for every character that is acceptable. The bits are counted fro significant end of each byte.

The reason for having both OP_CLASS and OP_NCLASS is so that, in UTF subject characters with values greater than 256 can be handled corre OP_CLASS they don't match, whereas for OP_NCLASS they do.

For classes containing characters with values > 255, OP_XCLASS is us optionally uses a bit map (if any characters lie within it), followe of pairs and single characters. There is a flag character than indic whether it's a positive or a negative class.

Back references

OP_REF is followed by two bytes containing the reference number.

Repeating character classes and back references

Single-character classes are handled specially (see above). This sec applies to OP_CLASS and OP_REF. In both cases, the repeat informatic the base item. The matching code looks at the following opcode to se one of

OP_CRSTAR OP_CRMINSTAR OP_CRPLUS OP_CRMINPLUS OP_CRQUERY OP_CRMINQUERY OP_CRRANGE OP_CRMINRANGE

All but the last two are just single-byte items. The others are foll four bytes of data, comprising the minimum and maximum repeat counts no special possessive opcodes for these repeats; a possessive repeat compiled into an atomic group.

Brackets and alternation

A pair of non-capturing (round) brackets is wrapped round each expre compile time, so alternation always happens in the context of bracke

[Note for North Americans: "bracket" to some English speakers, inclu myself, can be round, square, curly, or pointy. Hence this usage.]

Non-capturing brackets use the opcode OP_BRA. Originally PCRE was li capturing brackets and it used a different opcode for each one. From 3.5, the limit was removed by putting the bracket number into the da higher-numbered brackets. From release 7.0 all capturing brackets ar this way, using the single opcode OP_CBRA.

A bracket opcode is followed by LINK_SIZE bytes which give the offse next alternative OP_ALT or, if there aren't any branches, to the mat OP_KET opcode. Each OP_ALT is followed by LINK_SIZE bytes giving the the next one, or to the OP_KET opcode. For capturing brackets, the b number immediately follows the offset, always as a 2-byte item.

OP_KET is used for subpatterns that do not repeat indefinitely, whil OP_KETRMIN and OP_KETRMAX are used for indefinite repetitions, minim maximally respectively. All three are followed by LINK_SIZE bytes gi positive number) the offset back to the matching bracket opcode.

If a subpattern is quantified such that it is permitted to match zer is preceded by one of OP_BRAZERO or OP_BRAMINZERO. These are singleopcodes which tell the matcher that skipping this subpattern entirel valid branch.

A subpattern with an indefinite maximum repetition is replicated in compiled data its minimum number of times (or once with OP_BRAZERO i minimum is zero), with the final copy terminating with OP_KETRMIN or as appropriate.

A subpattern with a bounded maximum repetition is replicated in a ne fashion up to the maximum number of times, with OP_BRAZERO or OP_BRA before each replication after the minimum, so that, for example, (ab compiled as (abc)(abc)((abc)((abc)(abc)?)?)?, except that each brack has the same number.

When a repeated subpattern has an unbounded upper limit, it is check whether it could match an empty string. If this is the case, the opc final replication is changed to OP_SBRA or OP_SCBRA. This tells the that it needs to check for matching an empty string when it hits OP_ OP_KETRMAX, and if so, to break the loop.

Assertions

Forward assertions are just like other subpatterns, but starting wit the opcodes OP_ASSERT or OP_ASSERT_NOT. Backward assertions use the OP_ASSERTBACK and OP_ASSERTBACK_NOT, and the first opcode inside the is OP_REVERSE, followed by a two byte count of the number of charact back the pointer in the subject string. When operating in UTF-8 mode is a character count rather than a byte count. A separate count is p each alternative of a lookbehind assertion, allowing them to have di fixed lengths.

Once-only (atomic) subpatterns

These are also just like other subpatterns, but they start with the OP_ONCE. The check for matching an empty string in an unbounded repe handled entirely at runtime, so there is just this one opcode.

Conditional subpatterns

These are like other subpatterns, but they start with the opcode OP_OP_SCOND for one that might match an empty string in an unbounded re the condition is a back reference, this is stored at the start of th subpattern using the opcode OP_CREF followed by two bytes containing reference number. If the condition is "in recursion" (coded as "(?(R recursion of group x" (coded as "(?(Rx)"), the group number is store start of the subpattern using the opcode OP_RREF, and a value of zer whole pattern". For a DEFINE condition, just the single byte OP_DEF has no associated data). Otherwise, a conditional subpattern always one of the assertions.

Recursion

Recursion either matches the current regex, or some subexpression. T OP_RECURSE is followed by an value which is the offset to the starti from the start of the whole pattern. From release 6.5, OP_RECURSE is automatically wrapped inside OP_ONCE brackets (because otherwise som broke it). OP_RECURSE is also used for "subroutine" calls, even thou are not strictly a recursion.

Callout

- - - - - - -

OP_CALLOUT is followed by one byte of data that holds a callout numb range 0 to 254 for manual callouts, or 255 for an automatic callout. cases there follows a two-byte value giving the offset in the patter start of the following item, and another two-byte item giving the le next item.

Changing options

If any of the /i, /m, or /s options are changed within a pattern, an opcode is compiled, followed by one byte containing the new settings flags. If there are several alternatives, there is an occurrence of the start of all those following the first options change, to set ap options for the start of the alternative. Immediately after the end

group there is another such item to reset the flags to their previou change of flag right at the very start of the pattern can be handled at compile time, and so does not cause anything to be put into the c data.

Philip Hazel June 2007

Perl-compatible Regular Expressions (PCRE)

The HTML documentation for PCRE comprises the following pages:

<u>pcre</u>	Introductory page
<u>pcre-config</u>	Information about the installation configuration
<u>pcreapi</u>	PCRE's native API
<u>pcrebuild</u>	Options for building PCRE
<u>pcrecallout</u>	The <i>callout</i> facility
<u>pcrecompat</u>	Compability with Perl
<u>pcrecpp</u>	The C++ wrapper for the PCRE library
<u>pcregrep</u>	The pcregrep command
<u>pcrematching</u>	Discussion of the two matching algorithms
<u>pcrepartial</u>	Using PCRE for partial matching
<u>pcrepattern</u>	Specification of the regular expressions supported by PCRE
<u>pcreperform</u>	Some comments on performance
<u>pcreposix</u>	The POSIX API to the PCRE library
<u>pcreprecompile</u>	How to save and re-use compiled patterns
<u>pcresample</u>	Description of the sample program
<u>pcrestack</u>	Discussion of PCRE's stack usage
<u>pcretest</u>	The pcretest command for testing PCRE

There are also individual pages that summarize the interface for each function in the library:

<u>pcre_compile</u>	Compile a regular expression
<u>pcre_compile2</u>	Compile a regular expression (alternate interface)
<u>pcre_config</u>	Show build-time configuration options
pcre_copy_named_substring	Extract named substring into given buffer
<u>pcre_copy_substring</u>	Extract numbered substring into given buffer

pcre_dfa_exec

pcre_exec

pcre_free_substring pcre_free_substring_list pcre_fullinfo pcre_get_named_substring pcre_get_stringnumber pcre_get_substring_list pcre_get_substring_list pcre_info pcre_maketables pcre_refcount pcre_study pcre_version

Match a compiled pattern to a subject string (DFA algorithm; not Perl compatible) Match a compiled pattern to a subject string (Perl compatible) Free extracted substring Free list of extracted substrings Extract information about a pattern Extract named substring into new memory Convert captured string name to number Extract numbered substring into new memory Extract all substrings into new memory Obsolete information extraction function Build character tables in current locale Maintain reference count in compiled pattern Study a compiled pattern Return PCRE version and release date

pcre_compile man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

pcre *pcre_compile(const char *pattern, int options, const char **errptr, int *erroffset, const unsigned char *tableptr);

DESCRIPTION

This function compiles a regular expression into an internal form. It is the same as **pcre_compile2()**, except for the absence of the *errorcodeptr* argument. Its arguments are:

pattern	A zero-terminated string containing the regular expression to be compiled
antiona	• • •
options	Zero or more option bits
errptr	Where to put an error message
erroffset	Offset in pattern where error was found
tableptr	Pointer to character tables, or NULL to
	use the built-in default

The option bits are:

PCRE_ANCHORED PCRE_AUTO_CALLOUT	Force pattern anchoring Compile automatic callouts
PCRE_CASELESS	Do caseless matching
PCRE_DOLLAR_ENDONLY	<pre>\$ not to match newline at end</pre>
PCRE_DOTALL	. matches anything including NL
PCRE_DUPNAMES	Allow duplicate names for subpatterns
PCRE_EXTENDED	Ignore whitespace and # comments
PCRE_EXTRA	PCRE extra features
	(not much use currently)

PCRE_FIRSTLINE PCRE_MULTILINE PCRE_NEWLINE_ANY PCRE_NEWLINE_ANYCRLF PCRE_NEWLINE_CR PCRE_NEWLINE_CRLF	Force matching to be before newline ^ and \$ match newlines within data Recognize any Unicode newline sequence Recognize CR, LF, and CRLF as newline sequen Set CR as the newline sequence Set CRLF as the newline sequence
PCRE NEWLINE LF	Set LF as the newline sequence
PCRE_NO_AUTO_CAPTURE	
PCRE_UNGREEDY	Invert greediness of quantifiers
PCRE_UTF8	Run in UTF-8 mode
PCRE_NO_UTF8_CHECK	Do not check the pattern for UTF-8 validity (only relevant if PCRE_UTF8 is set)

PCRE must be built with UTF-8 support in order to use PCRE_UTF8 and PCRE_NO_UTF8_CHECK.

The yield of the function is a pointer to a private data structure that contains the compiled pattern, or NULL if an error was detected. Note that compiling regular expressions with one version of PCRE for use with a different version is not guaranteed to work and may cause crashes.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

Return to the <u>PCRE index page</u>.

pcre_compile2 man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

pcre *pcre_compile2(const char *pattern, int options, int *errorcodeptr, const char **errptr, int *erroffset, const unsigned char *tableptr);

DESCRIPTION

This function compiles a regular expression into an internal form. It is the same as **pcre_compile()**, except for the addition of the *errorcodeptr* argument. The arguments are:

pattern	A zero-terminated string containing the
	regular expression to be compiled
options	Zero or more option bits
errorcodeptr	Where to put an error code
errptr	Where to put an error message
erroffset	Offset in pattern where error was found
tableptr	Pointer to character tables, or NULL to
	use the built-in default

The option bits are:

PCRE_ANCHORED	Force pattern anchoring
PCRE_AUTO_CALLOUT	Compile automatic callouts
PCRE_CASELESS	Do caseless matching
PCRE_DOLLAR_ENDONLY	\$ not to match newline at end
PCRE_DOTALL	. matches anything including NL
PCRE_DUPNAMES	Allow duplicate names for subpatterns
PCRE_DUPNAMES	Allow duplicate names for subpatterns
PCRE_EXTENDED	Ignore whitespace and # comments
PCRE_EXTRA	PCRE extra features

	(not much use currently)
PCRE_FIRSTLINE	Force matching to be before newline
PCRE_MULTILINE	^ and \$ match newlines within data
PCRE_NEWLINE_ANY	Recognize any Unicode newline sequence
PCRE_NEWLINE_ANYCRLF	Recognize CR, LF, and CRLF as newline sequen
PCRE_NEWLINE_CR	Set CR as the newline sequence
PCRE_NEWLINE_CRLF	Set CRLF as the newline sequence
PCRE_NEWLINE_LF	Set LF as the newline sequence
PCRE_NO_AUTO_CAPTURE	Disable numbered capturing paren-
	theses (named ones available)
PCRE_UNGREEDY	Invert greediness of quantifiers
PCRE_UTF8	Run in UTF-8 mode
PCRE_NO_UTF8_CHECK	Do not check the pattern for UTF-8
	validity (only relevant if
	PCRE_UTF8 is set)

PCRE must be built with UTF-8 support in order to use PCRE_UTF8 and PCRE_NO_UTF8_CHECK.

The yield of the function is a pointer to a private data structure that contains the compiled pattern, or NULL if an error was detected. Note that compiling regular expressions with one version of PCRE for use with a different version is not guaranteed to work and may cause crashes.

There is a complete description of the PCRE native API in the **<u>pcreapi</u>** page and a description of the POSIX API in the **<u>pcreposix</u>** page.

Return to the <u>PCRE index page</u>.

pcre_config man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_config(int what, void *where);

DESCRIPTION

This function makes it possible for a client program to find out which optional features are available in the version of the PCRE library it is using. Its arguments are as follows:

what A code specifying what information is required where Points to where to put the data

The available codes are:

Internal link size Internal resource	
	denth limit
Value of the newli	
13 (0x000d)	for CR
10 (0x000a)	for LF
3338 (0x0d0a)	for CRLF
	for ANYCRLF
-1	for ANY
THRESHOLD	
Threshold of retur	n slots, above
which malloc() i	s used by
the POSIX API	,
	tation (1=stack 0=heap F-8 support (1=yes 0=n
	Internal resource ECURSION Internal recursion Value of the newli 13 (0x000d) 10 (0x000a) 3338 (0x0d0a) -2 -1 THRESHOLD Threshold of retur which malloc() i the POSIX API Recursion implemen

The function yields 0 on success or PCRE_ERROR_BADOPTION otherwise.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_copy_named_substring man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_copy_named_substring(const pcre *code, const char *subject, int *ovector, int stringcount, const char *stringname, char *buffer, int buffersize);

DESCRIPTION

This is a convenience function for extracting a captured substring, identified by name, into a given buffer. The arguments are:

code	Pattern that was successfully matched
subject	Subject that has been successfully matched
ovector	Offset vector that <pre>pcre_exec()</pre> used
stringcount	Value returned by pcre_exec()
stringname	Name of the required substring
buffer	Buffer to receive the string
buffersize	Size of buffer

The yield is the length of the substring, PCRE_ERROR_NOMEMORY if the buffer was too small, or PCRE_ERROR_NOSUBSTRING if the string name is invalid.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_copy_substring man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_copy_substring(const char *subject, int *ovector, int stringcount, int stringnumber, char *buffer, int buffersize);

DESCRIPTION

This is a convenience function for extracting a captured substring into a given buffer. The arguments are:

subject	Subject that has been successfully matched
ovector	Offset vector that pcre_exec() used
stringcount	Value returned by <pre>pcre_exec()</pre>
stringnumber	Number of the required substring
buffer	Buffer to receive the string
buffersize	Size of buffer

The yield is the length of the string, PCRE_ERROR_NOMEMORY if the buffer was too small, or PCRE_ERROR_NOSUBSTRING if the string number is invalid.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_dfa_exec man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_dfa_exec(const pcre *code, const pcre_extra *extra, const char *subject, int length, int startoffset, int options, int *ovector, int ovecsize, int *workspace, int wscount);

DESCRIPTION

This function matches a compiled regular expression against a given subject string, using an alternative matching algorithm that scans the subject string just once (*not* Perl-compatible). Note that the main, Perl-compatible, matching function is **pcre_exec()**. The arguments for this function are:

code	Points to the compiled pattern
extra	Points to an associated pcre_extra structure,
	or is NULL
subject	Points to the subject string
length	Length of the subject string, in bytes
startoffset	Offset in bytes in the subject at which to
	start matching
options	Option bits
ovector	Points to a vector of ints for result offsets
ovecsize	Number of elements in the vector
workspace	Points to a vector of ints used as working space
wscount	Number of elements in the vector

The options are:

PCRE_ANCHORED	Match only at	the first position
PCRE_NEWLINE_ANY	Recognize any	Unicode newline sequence

PCRE_NEWLINE_ANYCRLF Recognize CR, LF, and CRLF as newline sequen Set CR as the newline sequence PCRE NEWLINE CR Set CRLF as the newline sequence PCRE_NEWLINE_CRLF PCRE NEWLINE LF Set LF as the newline sequence PCRE NOTBOL Subject is not the beginning of a line PCRE NOTEOL Subject is not the end of a line PCRE NOTEMPTY An empty string is not a valid match PCRE_NO_UTF8_CHECK Do not check the subject for UTF-8 validity (only relevant if PCRE_UTF8 was set at compile time) Return PCRE_ERROR_PARTIAL for a partial match PCRE_PARTIAL PCRE_DFA_SHORTEST Return only the shortest match This is a restart after a partial match PCRE DFA RESTART

There are restrictions on what may appear in a pattern when using this matching function. Details are given in the **pcrematching** documentation.

A **pcre_extra** structure contains the following fields:

flags Bits indicating which fields are set study_data Opaque data from pcre_study() match_limit Limit on internal resource use match_limit_recursion Limit on internal recursion depth callout_data Opaque data passed back to callouts tables Points to character tables or is NULL

The flag bits are PCRE_EXTRA_STUDY_DATA, PCRE_EXTRA_MATCH_LIMIT, PCRE_EXTRA_MATCH_LIMIT_RECURSION, PCRE_EXTRA_CALLOUT_DATA, and PCRE_EXTRA_TABLES. For this matching function, the *match_limit* and *match_limit_recursion* fields are not used, and must not be set.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_exec man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_exec(const pcre *code, const pcre_extra *extra, const char *subject, int length, int startoffset, int options, int *ovector, int ovecsize);

DESCRIPTION

This function matches a compiled regular expression against a given subject string, using a matching algorithm that is similar to Perl's. It returns offsets to captured substrings. Its arguments are:

code	Points to the compiled pattern
extra	Points to an associated pcre_extra structure,
	or is NULL
subject	Points to the subject string
length	Length of the subject string, in bytes
startoffset	Offset in bytes in the subject at which to
	start matching
options	Option bits
ovector	Points to a vector of ints for result offsets
ovecsize	Number of elements in the vector (a multiple of 3)

The options are:

PCRE_ANCHOREDMatch only at the first positionPCRE_NEWLINE_ANYRecognize any Unicode newline sequencePCRE_NEWLINE_ANYCRLFRecognize CR, LF, and CRLF as newline sequenPCRE_NEWLINE_CRSet CR as the newline sequencePCRE_NEWLINE_CRLFSet CRLF as the newline sequencePCRE_NEWLINE_LFSet LF as the newline sequence

PCRE_NOTBOL	Subject is not the beginning of a line	
PCRE_NOTEOL	Subject is not the end of a line	
PCRE_NOTEMPTY	An empty string is not a valid match	
PCRE_NO_UTF8_CHECK	Do not check the subject for UTF-8	
validity (only relevant if PCRE_UTF8		
	was set at compile time)	
PCRE_PARTIAL	Return PCRE_ERROR_PARTIAL for a partial match	

There are restrictions on what may appear in a pattern when partial matching is requested. For details, see the **pcrepartial** page.

A **pcre_extra** structure contains the following fields:

flags Bits indicating which fields are set study_data Opaque data from pcre_study() match_limit Limit on internal resource use match_limit_recursion Limit on internal recursion depth callout_data Opaque data passed back to callouts tables Points to character tables or is NULL

The flag bits are PCRE_EXTRA_STUDY_DATA, PCRE_EXTRA_MATCH_LIMIT, PCRE_EXTRA_MATCH_LIMIT_RECURSION, PCRE_EXTRA_CALLOUT_DATA, and PCRE_EXTRA_TABLES.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_free_substring man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

void pcre_free_substring(const char *stringptr);

DESCRIPTION

This is a convenience function for freeing the store obtained by a previous call to **pcre_get_substring()** or **pcre_get_named_substring()**. Its only argument is a pointer to the string.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_free_substring_list man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

void pcre_free_substring_list(const char **stringptr);

DESCRIPTION

This is a convenience function for freeing the store obtained by a previous call to **pcre_get_substring_list()**. Its only argument is a pointer to the list of string pointers.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_fullinfo man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_fullinfo(const pcre *code, const pcre_extra *extra, int what, void *where);

DESCRIPTION

This function returns information about a compiled pattern. Its arguments are:

code	Compiled regular expression
extra	Result of <pre>pcre_study()</pre> or <pre>NULL</pre>
what	What information is required
where	Where to put the information

The following information is available:

PCRE_INFO_BACKREFMAX PCRE_INFO_CAPTURECOUNT PCRE_INFO_DEFAULT_TABLES PCRE_INFO_FIRSTBYTE	Number of highest back reference Number of capturing subpatterns Pointer to default tables Fixed first byte for a match, or -1 for start of string or after newline, or -2 otherwise
PCRE_INFO_FIRSTTABLE	Table of first bytes (after studying)
PCRE_INFO_JCHANGED	Return 1 if (?J) was used
PCRE_INFO_LASTLITERAL	Literal last byte required
PCRE_INFO_NAMECOUNT	Number of named subpatterns
PCRE_INFO_NAMEENTRYSIZE	Size of name table entry
PCRE_INFO_NAMETABLE	Pointer to name table
PCRE_INFO_OKPARTIAL	Return 1 if partial matching can be trie
PCRE_INFO_OPTIONS	Option bits used for compilation

PCRE_INFO_SIZE	Size of compiled pattern
PCRE_INF0_STUDYSIZE	Size of study data

The yield of the function is zero on success or:

PCRE_ERROR_NULL	the	argument <i>code</i> was NULL
	the	argument where was NULL
PCRE_ERROR_BADMAGIC	the	"magic number" was not found
PCRE_ERROR_BADOPTION	the	value of <i>what</i> was invalid

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_get_named_substring man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_get_named_substring(const pcre *code, const char *subject, int *ovector, int stringcount, const char *stringname, const char **stringptr);

DESCRIPTION

This is a convenience function for extracting a captured substring by name. The arguments are:

code	Compiled pattern
subject	Subject that has been successfully matched
ovector	Offset vector that pcre_exec() used
stringcount	Value returned by pcre_exec()
stringname	Name of the required substring
stringptr	Where to put the string pointer

The memory in which the substring is placed is obtained by calling **pcre_malloc()**. The convenience function **pcre_free_substring()** can be used to free it when it is no longer needed. The yield of the function is the length of the extracted substring, PCRE_ERROR_NOMEMORY if sufficient memory could not be obtained, or PCRE_ERROR_NOSUBSTRING if the string name is invalid.

There is a complete description of the PCRE native API in the **<u>pcreapi</u>** page and a description of the POSIX API in the **<u>pcreposix</u>** page.

pcre_get_stringnumber man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_get_stringnumber(const pcre *code, const char *name);

DESCRIPTION

This convenience function finds the number of a named substring capturing parenthesis in a compiled pattern. Its arguments are:

code Compiled regular expression
name Name whose number is required

The yield of the function is the number of the parenthesis if the name is found, or PCRE_ERROR_NOSUBSTRING otherwise. When duplicate names are allowed (PCRE_DUPNAMES is set), it is not defined which of the numbers is returned by **pcre_get_stringnumber()**. You can obtain the complete list by calling **pcre_get_stringtable_entries()**.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_get_substring man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_get_substring(const char *subject, int *ovector, int stringcount, int stringnumber, const char **stringptr);

DESCRIPTION

This is a convenience function for extracting a captured substring. The arguments are:

subject	Subject that has been successfully matched
ovector	Offset vector that <pre>pcre_exec()</pre> used
stringcount	Value returned by pcre_exec()
stringnumber	Number of the required substring
stringptr	Where to put the string pointer

The memory in which the substring is placed is obtained by calling **pcre_malloc()**. The convenience function **pcre_free_substring()** can be used to free it when it is no longer needed. The yield of the function is the length of the substring, PCRE_ERROR_NOMEMORY if sufficient memory could not be obtained, or PCRE_ERROR_NOSUBSTRING if the string number is invalid.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_get_substring_list man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_get_substring_list(const char *subject, int *ovector, int stringcount, const char ***listptr);

DESCRIPTION

This is a convenience function for extracting a list of all the captured substrings. The arguments are:

subject	Subject that has been successfully matched
ovector	Offset vector that pcre_exec used
stringcount	Value returned by pcre_exec
listptr	Where to put a pointer to the list

The memory in which the substrings and the list are placed is obtained by calling **pcre_malloc()**. The convenience function **pcre_free_substring_list()** can be used to free it when it is no longer needed. A pointer to a list of pointers is put in the variable whose address is in *listptr*. The list is terminated by a NULL pointer. The yield of the function is zero on success or PCRE_ERROR_NOMEMORY if sufficient memory could not be obtained.

There is a complete description of the PCRE native API in the **<u>pcreapi</u>** page and a description of the POSIX API in the **<u>pcreposix</u>** page.

pcre_info man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_info(const pcre *code, int *optptr, int *firstcharptr);

DESCRIPTION

This function is obsolete. You should be using **pcre_fullinfo()** instead.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre_maketables man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

const unsigned char *pcre_maketables(void);

DESCRIPTION

This function builds a set of character tables for character values less than 256. These can be passed to **pcre_compile()** to override PCRE's internal, built-in tables (which were made by **pcre_maketables()** when PCRE was compiled). You might want to do this if you are using a non-standard locale. The function yields a pointer to the tables.

There is a complete description of the PCRE native API in the **<u>pcreapi</u>** page and a description of the POSIX API in the **<u>pcreposix</u>** page.

pcre_refcount man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

int pcre_refcount(pcre *code, int adjust);

DESCRIPTION

This function is used to maintain a reference count inside a data block that contains a compiled pattern. Its arguments are:

code	Compiled regular expression
adjust	Adjustment to reference value

The yield of the function is the adjusted reference value, which is constrained to lie between 0 and 65535.

There is a complete description of the PCRE native API in the **<u>pcreapi</u>** page and a description of the POSIX API in the **<u>pcreposix</u>** page.

pcre_study man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

pcre_extra *pcre_study(const pcre *code, int options, const char **errptr);

DESCRIPTION

This function studies a compiled pattern, to see if additional information can be extracted that might speed up matching. Its arguments are:

code	A compiled regular expression
options	Options for <pre>pcre_study()</pre>
errptr	Where to put an error message

If the function succeeds, it returns a value that can be passed to **pcre_exec()** via its *extra* argument.

If the function returns NULL, either it could not find any additional information, or there was an error. You can tell the difference by looking at the error value. It is NULL in first case.

There are currently no options defined; the value of the second argument should always be zero.

There is a complete description of the PCRE native API in the **<u>pcreapi</u>** page and a description of the POSIX API in the **<u>pcreposix</u>** page.

pcre_version man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

SYNOPSIS

#include <pcre.h>

char *pcre_version(void);

DESCRIPTION

This function returns a character string that gives the version number of the PCRE library and the date of its release.

There is a complete description of the PCRE native API in the **pcreapi** page and a description of the POSIX API in the **pcreposix** page.

pcre man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- INTRODUCTION
- USER DOCUMENTATION
- <u>LIMITATIONS</u>
- <u>UTF-8 AND UNICODE PROPERTY SUPPORT</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

INTRODUCTION

The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl, with just a few differences. (Certain features that appeared in Python and PCRE before they appeared in Perl are also available using the Python syntax.)

The current implementation of PCRE (release 7.x) corresponds approximately with Perl 5.10, including support for UTF-8 encoded strings and Unicode general category properties. However, UTF-8 and Unicode support has to be explicitly enabled; it is not the default. The Unicode tables correspond to Unicode release 5.0.0.

In addition to the Perl-compatible matching function, PCRE contains an alternative matching function that matches the same compiled patterns in a different way. In certain circumstances, the alternative function has some advantages. For a discussion of the two matching algorithms, see the **pcrematching** page.

PCRE is written in C and released as a C library. A number of people have written wrappers and interfaces of various kinds. In particular, Google Inc. have

provided a comprehensive C++ wrapper. This is now included as part of the PCRE distribution. The **pcrecpp** page has details of this interface. Other people's contributions can be found in the *Contrib* directory at the primary FTP site, which is: <u>ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre</u>

Details of exactly which Perl regular expression features are and are not supported by PCRE are given in separate documents. See the **pcrepattern** and **pcrecompat** pages.

Some features of PCRE can be included, excluded, or changed when the library is built. The <u>pcre_config()</u> function makes it possible for a client to discover which features are available. The features themselves are described in the <u>pcrebuild</u> page. Documentation about building PCRE for various operating systems can be found in the **README** file in the source distribution.

The library contains a number of undocumented internal functions and data tables that are used by more than one of the exported external functions, but which are not intended for use by external callers. Their names all begin with "_pcre_", which hopefully will not provoke any name clashes. In some environments, it is possible to control which external symbols are exported when a shared library is built, and in these cases the undocumented symbols are not exported.

USER DOCUMENTATION

The user documentation for PCRE comprises a number of different sections. In the "man" format, each of these is a separate "man page". In the HTML format, each is a separate page, linked from the index page. In the plain text format, all the sections are concatenated, for ease of searching. The sections are as follows:

pcre pcre-config pcreapi	this document show PCRE installation configuration information details of PCRE's native C API
pcrebuild	options for building PCRE
pcrecallout	details of the callout feature
pcrecompat	discussion of Perl compatibility
pcrecpp	details of the C++ wrapper
pcregrep	description of the pcregrep command
pcrematching	discussion of the two matching algorithms
pcrepartial	details of the partial matching facility
pcrepattern	syntax and semantics of supported regular expres

pcreperform	discussion of performance issues
pcreposix	the POSIX-compatible C API
pcreprecompile	details of saving and re-using precompiled patte
pcresample	discussion of the sample program
pcrestack	discussion of stack usage
pcretest	description of the pcretest testing command

In addition, in the "man" and HTML formats, there is a short page for each C library function, listing its arguments and results.

LIMITATIONS

There are some size limitations in PCRE but it is hoped that they will never in practice be relevant.

The maximum length of a compiled pattern is 65539 (sic) bytes if PCRE is compiled with the default internal linkage size of 2. If you want to process regular expressions that are truly enormous, you can compile PCRE with an internal linkage size of 3 or 4 (see the **README** file in the source distribution and the **pcrebuild** documentation for details). In these cases the limit is substantially larger. However, the speed of execution is slower.

All values in repeating quantifiers must be less than 65536. The maximum compiled length of subpattern with an explicit repeat count is 30000 bytes. The maximum number of capturing subpatterns is 65535.

There is no limit to the number of parenthesized subpatterns, but there can be no more than 65535 capturing subpatterns.

The maximum length of name for a named subpattern is 32 characters, and the maximum number of named subpatterns is 10000.

The maximum length of a subject string is the largest positive number that an integer variable can hold. However, when using the traditional matching function, PCRE uses recursion to handle subpatterns and indefinite repetition. This means that the available stack space may limit the size of a subject string that can be processed by certain patterns. For a discussion of stack issues, see the **pcrestack** documentation.

UTF-8 AND UNICODE PROPERTY SUPPORT

From release 3.3, PCRE has had some support for character strings encoded in the UTF-8 format. For release 4.0 this was greatly extended to cover most common requirements, and in release 5.0 additional support for Unicode general category properties was added.

In order process UTF-8 strings, you must build PCRE to include UTF-8 support in the code, and, in addition, you must call **pcre_compile()** with the PCRE_UTF8 option flag. When you do this, both the pattern and any subject strings that are matched against it are treated as UTF-8 strings instead of just strings of bytes.

If you compile PCRE with UTF-8 support, but do not use it at run time, the library will be a bit bigger, but the additional run time overhead is limited to testing the PCRE_UTF8 flag occasionally, so should not be very big.

If PCRE is built with Unicode character property support (which implies UTF-8 support), the escape sequences $p{..}, P{..}$, and X are supported. The available properties that can be tested are limited to the general category properties such as Lu for an upper case letter or Nd for a decimal number, the Unicode script names such as Arabic or Han, and the derived properties Any and L&. A full list is given in the **pcrepattern** documentation. Only the short names for properties are supported. For example, $p{L}$ matches a letter. Its Perl synonym, $p{Letter}$, is not supported. Furthermore, in Perl, many properties may optionally be prefixed by "Is", for compatibility with Perl 5.6. PCRE does not support this.

The following comments apply when PCRE is running in UTF-8 mode:

1. When you set the PCRE_UTF8 flag, the strings passed as patterns and subjects are checked for validity on entry to the relevant functions. If an invalid UTF-8 string is passed, an error return is given. In some situations, you may already know that your strings are valid, and therefore want to skip these checks in order to improve performance. If you set the PCRE_NO_UTF8_CHECK flag at compile time or at run time, PCRE assumes that the pattern or subject it is given (respectively) contains only valid UTF-8 codes. In this case, it does not diagnose an invalid UTF-8 string. If you pass an invalid UTF-8 string to PCRE when PCRE_NO_UTF8_CHECK is set, the results are undefined. Your program may crash.

2. An unbraced hexadecimal escape sequence (such as \xb3) matches a two-byte UTF-8 character if the value is greater than 127.

3. Octal numbers up to \777 are recognized, and match two-byte UTF-8 characters for values greater than \177.

4. Repeat quantifiers apply to complete UTF-8 characters, not to individual bytes, for example: $x{100}{3}$.

5. The dot metacharacter matches one UTF-8 character instead of a single byte.

6. The escape sequence \C can be used to match a single byte in UTF-8 mode, but its use can lead to some strange effects. This facility is not available in the alternative matching function, **pcre_dfa_exec()**.

7. The character escapes \b, \B, \d, \D, \s, \S, \w, and \W correctly test characters of any code value, but the characters that PCRE recognizes as digits, spaces, or word characters remain the same set as before, all with values less than 256. This remains true even when PCRE includes Unicode property support, because to do otherwise would slow down PCRE in many common cases. If you really want to test for a wider sense of, say, "digit", you must use Unicode property tests such as $p{Nd}$.

8. Similarly, characters that match the POSIX named character classes are all low-valued characters.

9. However, the Perl 5.10 horizontal and vertical whitespace matching escapes (\h, \H, \v, and \V) do match all the appropriate Unicode characters.

10. Case-insensitive matching applies only to characters whose values are less than 128, unless PCRE is built with Unicode property support. Even when Unicode property support is available, PCRE still uses its own character tables when checking the case of low-valued characters, so as not to degrade performance. The Unicode property information is used only for characters with higher values. Even when Unicode property support is available, PCRE supports case-insensitive matching only when there is a one-to-one mapping between a letter's cases. There are a small number of many-to-one mappings in Unicode; these are not supported by PCRE.

<u>AUTHOR</u>

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

Putting an actual email address here seems to have been a spam magnet, so I've taken it away. If you want to email me, use my two initials, followed by the two digits 10, at the domain cam.ac.uk.

REVISION

Last updated: 13 June 2007 Copyright © 1997-2007 University of Cambridge.

pcreapi man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- <u>PCRE NATIVE API</u>
- <u>PCRE API OVERVIEW</u>
- <u>NEWLINES</u>
- <u>MULTITHREADING</u>
- <u>SAVING PRECOMPILED PATTERNS FOR LATER USE</u>
- <u>CHECKING BUILD-TIME OPTIONS</u>
- <u>COMPILING A PATTERN</u>
- <u>COMPILATION ERROR CODES</u>
- <u>STUDYING A PATTERN</u>
- LOCALE SUPPORT
- INFORMATION ABOUT A PATTERN
- OBSOLETE INFO FUNCTION
- <u>REFERENCE COUNTS</u>
- MATCHING A PATTERN: THE TRADITIONAL FUNCTION
- <u>EXTRACTING CAPTURED SUBSTRINGS BY NUMBER</u>
- EXTRACTING CAPTURED SUBSTRINGS BY NAME
- DUPLICATE SUBPATTERN NAMES
- FINDING ALL POSSIBLE MATCHES
- MATCHING A PATTERN: THE ALTERNATIVE FUNCTION
- <u>SEE ALSO</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

PCRE NATIVE API

#include <pcre.h>

pcre *pcre_compile(const char *pattern, int options, const char **errptr, int

*erroffset, const unsigned char *tableptr);

pcre *pcre_compile2(const char *pattern, int options, int *errorcodeptr, const char **errptr, int *erroffset, const unsigned char *tableptr);

pcre_extra *pcre_study(const pcre *code, int options, const char **errptr);

int pcre_exec(const pcre *code, const pcre_extra *extra, const char *subject, int length, int startoffset, int options, int *ovector, int ovecsize);

int pcre_dfa_exec(const pcre *code, const pcre_extra *extra, const char *subject, int length, int startoffset, int options, int *ovector, int ovecsize, int *workspace, int wscount);

int pcre_copy_named_substring(const pcre *code, const char *subject, int *ovector, int stringcount, const char *stringname, char *buffer, int buffersize);

int pcre_copy_substring(const char *subject, int *ovector, int stringcount, int stringnumber, char *buffer, int buffersize);

int pcre_get_named_substring(const pcre *code, const char *subject, int *ovector, int stringcount, const char *stringname, const char **stringptr);

int pcre_get_stringnumber(const pcre *code, const char *name);

int pcre_get_stringtable_entries(const pcre *code, const char *name, char
**first, char **last);

int pcre_get_substring(const char *subject, int *ovector, int stringcount, int stringnumber, const char **stringptr);

int pcre_get_substring_list(const char *subject, int *ovector, int stringcount, const char ***listptr);

void pcre_free_substring(const char *stringptr);

void pcre_free_substring_list(const char **stringptr);

const unsigned char *pcre_maketables(void);

int pcre_fullinfo(const pcre *code, const pcre_extra *extra, int what, void *where);

int pcre_info(const pcre *code, int *optptr, int *firstcharptr);

int pcre_refcount(pcre *code, int adjust);

int pcre_config(int what, void *where);

char *pcre_version(void);

void *(*pcre_malloc)(size_t);

void (*pcre_free)(void *);

void *(*pcre_stack_malloc)(size_t);

void (*pcre_stack_free)(void *);

int (*pcre_callout)(pcre_callout_block *);

PCRE API OVERVIEW

PCRE has its own native API, which is described in this document. There are also some wrapper functions that correspond to the POSIX regular expression API. These are described in the **pcreposix** documentation. Both of these APIs define a set of C function calls. A C++ wrapper is distributed with PCRE. It is documented in the **pcrecpp** page.

The native API C function prototypes are defined in the header file **pcre.h**, and on Unix systems the library itself is called **libpcre**. It can normally be accessed by adding **-lpcre** to the command for linking an application that uses PCRE. The header file defines the macros PCRE_MAJOR and PCRE_MINOR to contain the major and minor release numbers for the library. Applications can use these to include support for different releases of PCRE.

The functions **pcre_compile()**, **pcre_compile2()**, **pcre_study()**, and **pcre_exec()** are used for compiling and matching regular expressions in a Perl-compatible manner. A sample program that demonstrates the simplest way of using them is

provided in the file called *pcredemo.c* in the source distribution. The **pcresample** documentation describes how to run it.

A second matching function, **pcre_dfa_exec()**, which is not Perl-compatible, is also provided. This uses a different algorithm for the matching. The alternative algorithm finds all possible matches (at a given point in the subject), and scans the subject just once. However, this algorithm does not return captured substrings. A description of the two matching algorithms and their advantages and disadvantages is given in the **pcrematching** documentation.

In addition to the main compiling and matching functions, there are convenience functions for extracting captured substrings from a subject string that is matched by **pcre_exec()**. They are:

```
pcre_copy_substring()
pcre_copy_named_substring()
pcre_get_substring()
pcre_get_named_substring()
pcre_get_substring_list()
pcre_get_stringnumber()
pcre_get_stringtable_entries()
```

pcre_free_substring() and pcre_free_substring_list() are also provided, to free the memory used for extracted strings.

The function **pcre_maketables()** is used to build a set of character tables in the current locale for passing to **pcre_compile()**, **pcre_exec()**, or **pcre_dfa_exec()**. This is an optional facility that is provided for specialist use. Most commonly, no special tables are passed, in which case internal tables that are generated when PCRE is built are used.

The function **pcre_fullinfo()** is used to find out information about a compiled pattern; **pcre_info()** is an obsolete version that returns only some of the available information, but is retained for backwards compatibility. The function **pcre_version()** returns a pointer to a string containing the version of PCRE and its date of release.

The function **pcre_refcount()** maintains a reference count in a data block containing a compiled pattern. This is provided for the benefit of object-oriented applications.

The global variables **pcre_malloc** and **pcre_free** initially contain the entry points of the standard **malloc()** and **free()** functions, respectively. PCRE calls the memory management functions via these variables, so a calling program can replace them if it wishes to intercept the calls. This should be done before calling any PCRE functions.

The global variables **pcre_stack_malloc** and **pcre_stack_free** are also indirections to memory management functions. These special functions are used only when PCRE is compiled to use the heap for remembering data, instead of recursive function calls, when running the **pcre_exec()** function. See the **pcrebuild** documentation for details of how to do this. It is a non-standard way of building PCRE, for use in environments that have limited stacks. Because of the greater use of memory management, it runs more slowly. Separate functions are provided so that special-purpose external code can be used for this case. When used, these functions are always called in a stack-like manner (last obtained, first freed), and always for memory blocks of the same size. There is a discussion about PCRE's stack usage in the **pcrestack** documentation.

The global variable **pcre_callout** initially contains NULL. It can be set by the caller to a "callout" function, which PCRE will then call at specified points during a matching operation. Details are given in the **pcrecallout** documentation.

NEWLINES

PCRE supports five different conventions for indicating line breaks in strings: a single CR (carriage return) character, a single LF (linefeed) character, the two-character sequence CRLF, any of the three preceding, or any Unicode newline sequence. The Unicode newline sequences are the three just mentioned, plus the single characters VT (vertical tab, U+000B), FF (formfeed, U+000C), NEL (next line, U+0085), LS (line separator, U+2028), and PS (paragraph separator, U+2029).

Each of the first three conventions is used by at least one operating system as its standard newline sequence. When PCRE is built, a default can be specified. The default default is LF, which is the Unix standard. When PCRE is run, the default can be overridden, either when a pattern is compiled, or when it is matched.

In the PCRE documentation the word "newline" is used to mean "the character or pair of characters that indicate a line break". The choice of newline convention affects the handling of the dot, circumflex, and dollar metacharacters, the handling of #-comments in /x mode, and, when CRLF is a recognized line ending sequence, the match position advancement for a non-anchored pattern. The choice of newline convention does not affect the interpretation of the \n or \r escape sequences.

MULTITHREADING

The PCRE functions can be used in multi-threading applications, with the proviso that the memory management functions pointed to by **pcre_malloc**, **pcre_free**, **pcre_stack_malloc**, and **pcre_stack_free**, and the callout function pointed to by **pcre_callout**, are shared by all threads.

The compiled form of a regular expression is not altered during matching, so the same compiled pattern can safely be used by several threads at once.

SAVING PRECOMPILED PATTERNS FOR LATER USE

The compiled form of a regular expression can be saved and re-used at a later time, possibly by a different program, and even on a host other than the one on which it was compiled. Details are given in the **pcreprecompile** documentation. However, compiling a regular expression with one version of PCRE for use with a different version is not guaranteed to work and may cause crashes.

CHECKING BUILD-TIME OPTIONS

int pcre_config(int what, void *where);

The function **pcre_config()** makes it possible for a PCRE client to discover which optional features have been compiled into the PCRE library. The **pcrebuild** documentation has more details about these optional features.

The first argument for **pcre_config()** is an integer, specifying which information is required; the second argument is a pointer to a variable into which the information is placed. The following information is available:

PCRE_CONFIG_UTF8

The output is an integer that is set to one if UTF-8 support is available; otherwise it is set to zero.

PCRE_CONFIG_UNICODE_PROPERTIES

The output is an integer that is set to one if support for Unicode character properties is available; otherwise it is set to zero.

PCRE_CONFIG_NEWLINE

The output is an integer whose value specifies the default character sequence that is recognized as meaning "newline". The four values that are supported are: 10 for LF, 13 for CR, 3338 for CRLF, -2 for ANYCRLF, and -1 for ANY. The default should normally be the standard sequence for your operating system.

```
PCRE_CONFIG_LINK_SIZE
```

The output is an integer that contains the number of bytes used for internal linkage in compiled regular expressions. The value is 2, 3, or 4. Larger values allow larger regular expressions to be compiled, at the expense of slower matching. The default value of 2 is sufficient for all but the most massive patterns, since it allows the compiled pattern to be up to 64K in size.

PCRE_CONFIG_POSIX_MALLOC_THRESHOLD

The output is an integer that contains the threshold above which the POSIX interface uses **malloc()** for output vectors. Further details are given in the **pcreposix** documentation.

PCRE_CONFIG_MATCH_LIMIT

The output is an integer that gives the default limit for the number of internal matching function calls in a **pcre_exec()** execution. Further details are given with **pcre_exec()** below.

PCRE_CONFIG_MATCH_LIMIT_RECURSION

The output is an integer that gives the default limit for the depth of recursion when calling the internal matching function in a **pcre_exec()** execution. Further details are given with **pcre_exec()** below.

```
PCRE_CONFIG_STACKRECURSE
```

The output is an integer that is set to one if internal recursion when running **pcre_exec()** is implemented by recursive function calls that use the stack to remember their state. This is the usual way that PCRE is compiled. The output is zero if PCRE was compiled to use blocks of data on the heap instead of recursive function calls. In this case, **pcre_stack_malloc** and **pcre_stack_free** are called to manage memory blocks on the heap, thus avoiding the use of the stack.

COMPILING A PATTERN

pcre *pcre_compile(const char *pattern, int options, const char **errptr, int *erroffset, const unsigned char *tableptr); pcre *pcre_compile2(const char *pattern, int options, int *errorcodeptr, const char **errptr, int *erroffset, const unsigned char *tableptr);

Either of the functions **pcre_compile()** or **pcre_compile2()** can be called to compile a pattern into an internal form. The only difference between the two interfaces is that **pcre_compile2()** has an additional argument, *errorcodeptr*, via which a numerical error code can be returned.

The pattern is a C string terminated by a binary zero, and is passed in the *pattern* argument. A pointer to a single block of memory that is obtained via **pcre_malloc** is returned. This contains the compiled code and related data. The **pcre** type is defined for the returned block; this is a typedef for a structure whose contents are not externally defined. It is up to the caller to free the memory (via **pcre_free**) when it is no longer required.

Although the compiled code of a PCRE regex is relocatable, that is, it does not depend on memory location, the complete **pcre** data block is not fully relocatable, because it may contain a copy of the *tableptr* argument, which is an address (see below).

The *options* argument contains various bit settings that affect the compilation. It should be zero if no options are required. The available options are described below. Some of them, in particular, those that are compatible with Perl, can also be set and unset from within the pattern (see the detailed description in the **pcrepattern** documentation). For these options, the contents of the *options* argument specifies their initial settings at the start of compilation and execution.

The PCRE_ANCHORED and PCRE_NEWLINE_*xxx* options can be set at the time of matching as well as at compile time.

If *errptr* is NULL, **pcre_compile()** returns NULL immediately. Otherwise, if compilation of a pattern fails, **pcre_compile()** returns NULL, and sets the variable pointed to by *errptr* to point to a textual error message. This is a static string that is part of the library. You must not try to free it. The offset from the start of the pattern to the character where the error was discovered is placed in the variable pointed to by *erroffset*, which must not be NULL. If it is, an immediate error is given.

If **pcre_compile2()** is used instead of **pcre_compile()**, and the *errorcodeptr* argument is not NULL, a non-zero error code number is returned via this argument in the event of an error. This is in addition to the textual error message. Error codes and messages are listed below.

If the final argument, *tableptr*, is NULL, PCRE uses a default set of character tables that are built when PCRE is compiled, using the default C locale. Otherwise, *tableptr* must be an address that is the result of a call to **pcre_maketables()**. This value is stored with the compiled pattern, and used again by **pcre_exec()**, unless another table pointer is passed to it. For more discussion, see the section on locale support below.

This code fragment shows a typical straightforward call to **pcre_compile()**:

The following names for option bits are defined in the **pcre.h** header file:

PCRE_ANCHORED

If this bit is set, the pattern is forced to be "anchored", that is, it is constrained to match only at the first matching point in the string that is being searched (the "subject string"). This effect can also be achieved by appropriate constructs in

the pattern itself, which is the only way to do it in Perl.

PCRE_AUTO_CALLOUT

If this bit is set, **pcre_compile()** automatically inserts callout items, all with number 255, before each pattern item. For discussion of the callout facility, see the **pcrecallout** documentation.

PCRE_CASELESS

If this bit is set, letters in the pattern match both upper and lower case letters. It is equivalent to Perl's /i option, and it can be changed within a pattern by a (?i) option setting. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.

PCRE_DOLLAR_ENDONLY

If this bit is set, a dollar metacharacter in the pattern matches only at the end of the subject string. Without this option, a dollar also matches immediately before a newline at the end of the string (but not before any other newlines). The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set. There is no equivalent to this option in Perl, and no way to set it within a pattern.

PCRE_DOTALL

If this bit is set, a dot metacharater in the pattern matches all characters, including those that indicate newline. Without it, a dot does not match when the current position is at a newline. This option is equivalent to Perl's /s option, and it can be changed within a pattern by a (?s) option setting. A negative class such as [^a] always matches newline characters, independent of the setting of this option.

PCRE_DUPNAMES

If this bit is set, names used to identify capturing subpatterns need not be unique. This can be helpful for certain types of pattern when it is known that only one instance of the named subpattern can ever be matched. There are more details of named subpatterns below; see also the **pcrepattern** documentation.

PCRE_EXTENDED

If this bit is set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class. Whitespace does not include the VT character (code 11). In addition, characters between an unescaped # outside a character class and the next newline, inclusive, are also ignored. This is equivalent to Perl's /x option, and it can be changed within a pattern by a (?x) option setting.

This option makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence (?(which introduces a conditional subpattern.

PCRE_EXTRA

This option was invented in order to turn on additional functionality of PCRE that is incompatible with Perl, but it is currently of very little use. When set, any backslash in a pattern that is followed by a letter that has no special meaning causes an error, thus reserving these combinations for future expansion. By default, as in Perl, a backslash followed by a letter with no special meaning is treated as a literal. (Perl can, however, be persuaded to give a warning for this.) There are at present no other features controlled by this option. It can also be set by a (?X) option setting within a pattern.

PCRE_FIRSTLINE

If this option is set, an unanchored pattern is required to match before or at the first newline in the subject string, though the matched text may continue over the newline.

PCRE_MULTILINE

By default, PCRE treats the subject string as consisting of a single line of characters (even if it actually contains newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter (\$) matches only at the end of the string, or before a terminating newline (unless PCRE_DOLLAR_ENDONLY is set). This is the same as Perl.

When PCRE_MULTILINE it is set, the "start of line" and "end of line" constructs match immediately following or immediately before internal newlines in the subject string, respectively, as well as at the very start and end. This is

equivalent to Perl's /m option, and it can be changed within a pattern by a (?m) option setting. If there are no newlines in a subject string, or no occurrences of ^ or \$ in a pattern, setting PCRE_MULTILINE has no effect.

PCRE_NEWLINE_CR PCRE_NEWLINE_LF PCRE_NEWLINE_CRLF PCRE_NEWLINE_ANYCRLF PCRE_NEWLINE_ANY

These options override the default newline definition that was chosen when PCRE was built. Setting the first or the second specifies that a newline is indicated by a single character (CR or LF, respectively). Setting PCRE_NEWLINE_CRLF specifies that a newline is indicated by the twocharacter CRLF sequence. Setting PCRE_NEWLINE_ANYCRLF specifies that any of the three preceding sequences should be recognized. Setting PCRE_NEWLINE_ANY specifies that any Unicode newline sequence should be recognized. The Unicode newline sequences are the three just mentioned, plus the single characters VT (vertical tab, U+000B), FF (formfeed, U+000C), NEL (next line, U+0085), LS (line separator, U+2028), and PS (paragraph separator, U+2029). The last two are recognized only in UTF-8 mode.

The newline setting in the options word uses three bits that are treated as a number, giving eight possibilities. Currently only six are used (default plus the five values above). This means that if you set more than one newline option, the combination may or may not be sensible. For example, PCRE_NEWLINE_CR with PCRE_NEWLINE_LF is equivalent to PCRE_NEWLINE_CRLF, but other combinations may yield unused numbers and cause an error.

The only time that a line break is specially recognized when compiling a pattern is if PCRE_EXTENDED is set, and an unescaped # outside a character class is encountered. This indicates a comment that lasts until after the next line break sequence. In other circumstances, line break sequences are treated as literal data, except that in PCRE_EXTENDED mode, both CR and LF are treated as whitespace characters and are therefore ignored.

The newline option that is set at compile time becomes the default that is used for **pcre_exec()** and **pcre_dfa_exec()**, but it can be overridden.

PCRE_NO_AUTO_CAPTURE

If this option is set, it disables the use of numbered capturing parentheses in the pattern. Any opening parenthesis that is not followed by ? behaves as if it were followed by ?: but named parentheses can still be used for capturing (and they acquire numbers in the usual way). There is no equivalent of this option in Perl.

PCRE_UNGREEDY

This option inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by "?". It is not compatible with Perl. It can also be set by a (?U) option setting within the pattern.

PCRE_UTF8

This option causes PCRE to regard both the pattern and the subject as strings of UTF-8 characters instead of single-byte character strings. However, it is available only when PCRE is built to include UTF-8 support. If not, the use of this option provokes an error. Details of how this option changes the behaviour of PCRE are given in the <u>section on UTF-8 support</u> in the main <u>pcre</u> page.

PCRE_NO_UTF8_CHECK

When PCRE_UTF8 is set, the validity of the pattern as a UTF-8 string is automatically checked. If an invalid UTF-8 sequence of bytes is found, **pcre_compile()** returns an error. If you already know that your pattern is valid, and you want to skip this check for performance reasons, you can set the PCRE_NO_UTF8_CHECK option. When it is set, the effect of passing an invalid UTF-8 string as a pattern is undefined. It may cause your program to crash. Note that this option can also be passed to **pcre_exec()** and **pcre_dfa_exec()**, to suppress the UTF-8 validity checking of subject strings.

COMPILATION ERROR CODES

The following table lists the error codes than may be returned by **pcre_compile2()**, along with the error messages that may be returned by both compiling functions. As PCRE has developed, some error codes have fallen out of use. To avoid confusion, they have not been re-used.

- 0 no error
- 1 $\$ at end of pattern
- 2 \c at end of pattern
- 3 unrecognized character follows $\$
- 4 numbers out of order in {} quantifier

number too big in {} quantifier 5 6 missing terminating] for character class 7 invalid escape sequence in character class 8 range out of order in character class 9 nothing to repeat 10 [this code is not in use] 11 internal error: unexpected repeat 12 unrecognized character after (? 13 POSIX named classes are supported only within a class 14 missing) 15 reference to non-existent subpattern 16 erroffset passed as NULL 17 unknown option bit(s) set 18 missing) after comment 19 [this code is not in use] 20 regular expression too large 21 failed to get memory 22 unmatched parentheses 23 internal error: code overflow 24 unrecognized character after (?< 25 lookbehind assertion is not fixed length 26 malformed number or name after (?(27 conditional group contains more than two branches 28 assertion expected after (?(29 (?R or (?[+-]digits must be followed by) 30 unknown POSIX class name 31 POSIX collating elements are not supported 32 this version of PCRE is not compiled with PCRE_UTF8 support 33 [this code is not in use] 34 character value in $x{\dots}$ sequence is too large 35 invalid condition (?(0) \C not allowed in lookbehind assertion 36 37 PCRE does not support L, N, V, v, v, v38 number after (?C is > 255 39 closing) for (?C expected 40 recursive call could loop indefinitely 41 unrecognized character after (?P 42 syntax error in subpattern name (missing terminator) 43 two named subpatterns have the same name 44 invalid UTF-8 string support for P, p, and X has not been compiled 45 malformed \P or \p sequence 46 47 unknown property name after \P or \p 48 subpattern name is too long (maximum 32 characters) 49 too many named subpatterns (maximum 10,000) 50 repeated subpattern is too long octal value is greater than 377 (not in UTF-8 mode) 51 52 internal error: overran compiling workspace 53 internal error: previously-checked referenced subpattern not f 54 DEFINE group contains more than one branch

- 55 repeating a DEFINE group is not allowed
- 56 inconsistent NEWLINE options"
- 57 \g is not followed by a braced name or an optionally braced non-zero number
- 58 (?+ or (?- or (?(+ or (?(- must be followed by a non-zero numb

STUDYING A PATTERN

pcre_extra *pcre_study(const pcre *code, int options const char **errptr);

If a compiled pattern is going to be used several times, it is worth spending more time analyzing it in order to speed up the time taken for matching. The function **pcre_study()** takes a pointer to a compiled pattern as its first argument. If studying the pattern produces additional information that will help speed up matching, **pcre_study()** returns a pointer to a **pcre_extra** block, in which the *study_data* field points to the results of the study.

The returned value from **pcre_study()** can be passed directly to **pcre_exec()**. However, a **pcre_extra** block also contains other fields that can be set by the caller before the block is passed; these are described <u>below</u> in the section on matching a pattern.

If studying the pattern does not produce any additional information **pcre_study()** returns NULL. In that circumstance, if the calling program wants to pass any of the other fields to **pcre_exec()**, it must set up its own **pcre_extra** block.

The second argument of **pcre_study()** contains option bits. At present, no options are defined, and this argument should always be zero.

The third argument for **pcre_study()** is a pointer for an error message. If studying succeeds (even if no data is returned), the variable it points to is set to NULL. Otherwise it is set to point to a textual error message. This is a static string that is part of the library. You must not try to free it. You should test the error pointer for NULL after calling **pcre_study()**, to be sure that it has run successfully.

This is a typical call to **pcre_study**():

```
0, /* no options exist */
&error;); /* set to NULL or points to a message */
```

At present, studying a pattern is useful only for non-anchored patterns that do not have a single fixed starting character. A bitmap of possible starting bytes is created.

LOCALE SUPPORT

PCRE handles caseless matching, and determines whether characters are letters, digits, or whatever, by reference to a set of tables, indexed by character value. When running in UTF-8 mode, this applies only to characters with codes less than 128. Higher-valued codes never match escapes such as \w or \d, but can be tested with \p if PCRE is built with Unicode character property support. The use of locales with Unicode is discouraged. If you are handling characters with codes greater than 128, you should either use UTF-8 and Unicode, or use locales, but not try to mix the two.

PCRE contains an internal set of tables that are used when the final argument of **pcre_compile()** is NULL. These are sufficient for many applications. Normally, the internal tables recognize only ASCII characters. However, when PCRE is built, it is possible to cause the internal tables to be rebuilt in the default "C" locale of the local system, which may cause them to be different.

The internal tables can always be overridden by tables supplied by the application that calls PCRE. These may be created in a different locale from the default. As more and more applications change to using Unicode, the need for this locale support is expected to die away.

External tables are built by calling the **pcre_maketables()** function, which has no arguments, in the relevant locale. The result can then be passed to **pcre_compile()** or **pcre_exec()** as often as necessary. For example, to build and use tables that are appropriate for the French locale (where accented characters with values greater than 128 are treated as letters), the following code could be used:

```
setlocale(LC_CTYPE, "fr_FR");
tables = pcre_maketables();
re = pcre_compile(..., tables);
```

The locale name "fr_FR" is used on Linux and other Unix-like systems; if you are using Windows, the name for the French locale is "french".

When **pcre_maketables()** runs, the tables are built in memory that is obtained via **pcre_malloc**. It is the caller's responsibility to ensure that the memory containing the tables remains available for as long as it is needed.

The pointer that is passed to **pcre_compile()** is saved with the compiled pattern, and the same tables are used via this pointer by **pcre_study()** and normally also by **pcre_exec()**. Thus, by default, for any single pattern, compilation, studying and matching all happen in the same locale, but different patterns can be compiled in different locales.

It is possible to pass a table pointer or NULL (indicating the use of the internal tables) to **pcre_exec()**. Although not intended for this purpose, this facility could be used to match a pattern in a different locale from the one in which it was compiled. Passing table pointers at run time is discussed below in the section on matching a pattern.

INFORMATION ABOUT A PATTERN

int pcre_fullinfo(const pcre *code, const pcre_extra *extra, int what, void *where);

The **pcre_fullinfo()** function returns information about a compiled pattern. It replaces the obsolete **pcre_info()** function, which is nevertheless retained for backwards compability (and is documented below).

The first argument for **pcre_fullinfo()** is a pointer to the compiled pattern. The second argument is the result of **pcre_study()**, or NULL if the pattern was not studied. The third argument specifies which piece of information is required, and the fourth argument is a pointer to a variable to receive the data. The yield of the function is zero for success, or one of the following negative numbers:

PCRE_ERROR_NULL	the argument <i>code</i> was NULL
	the argument <i>where</i> was NULL
PCRE_ERROR_BADMAGIC	the "magic number" was not found
PCRE_ERROR_BADOPTION	the value of <i>what</i> was invalid

The "magic number" is placed at the start of each compiled pattern as an simple

check against passing an arbitrary memory pointer. Here is a typical call of **pcre_fullinfo()**, to obtain the length of the compiled pattern:

The possible values for the third argument are defined in **pcre.h**, and are as follows:

PCRE_INFO_BACKREFMAX

Return the number of the highest back reference in the pattern. The fourth argument should point to an **int** variable. Zero is returned if there are no back references.

PCRE_INFO_CAPTURECOUNT

Return the number of capturing subpatterns in the pattern. The fourth argument should point to an **int** variable.

PCRE_INFO_DEFAULT_TABLES

Return a pointer to the internal default character tables within PCRE. The fourth argument should point to an **unsigned char** * variable. This information call is provided for internal use by the **pcre_study()** function. External callers can cause PCRE to use its internal tables by passing a NULL table pointer.

PCRE_INFO_FIRSTBYTE

Return information about the first byte of any matched string, for a nonanchored pattern. The fourth argument should point to an **int** variable. (This option used to be called PCRE_INFO_FIRSTCHAR; the old name is still recognized for backwards compatibility.)

If there is a fixed first byte, for example, from a pattern such as (cat|cow|coyote), its value is returned. Otherwise, if either

(a) the pattern was compiled with the PCRE_MULTILINE option, and every branch starts with "^", or

(b) every branch of the pattern starts with ".*" and PCRE_DOTALL is not set (if it were set, the pattern would be anchored),

-1 is returned, indicating that the pattern matches only at the start of a subject string or after any newline within the string. Otherwise -2 is returned. For anchored patterns, -2 is returned.

PCRE_INFO_FIRSTTABLE

If the pattern was studied, and this resulted in the construction of a 256-bit table indicating a fixed set of bytes for the first byte in any matching string, a pointer to the table is returned. Otherwise NULL is returned. The fourth argument should point to an **unsigned char *** variable.

PCRE_INFO_JCHANGED

Return 1 if the (?J) option setting is used in the pattern, otherwise 0. The fourth argument should point to an **int** variable. The (?J) internal option setting changes the local PCRE_DUPNAMES option.

PCRE_INFO_LASTLITERAL

Return the value of the rightmost literal byte that must exist in any matched string, other than at its start, if such a byte has been recorded. The fourth argument should point to an **int** variable. If there is no such byte, -1 is returned. For anchored patterns, a last literal byte is recorded only if it follows something of variable length. For example, for the pattern //a\d+z\d+/ the returned value is "z", but for //a\dz\d/ the returned value is -1.

PCRE_INFO_NAMECOUNT PCRE_INFO_NAMEENTRYSIZE PCRE_INFO_NAMETABLE

PCRE supports the use of named as well as numbered capturing parentheses. The names are just an additional way of identifying the parentheses, which still acquire numbers. Several convenience functions such as

pcre_get_named_substring() are provided for extracting captured substrings by name. It is also possible to extract the data directly, by first converting the name to a number in order to access the correct pointers in the output vector (described with **pcre_exec()** below). To do the conversion, you need to use the name-to-number map, which is described by these three values.

The map consists of a number of fixed-size entries.

PCRE_INFO_NAMECOUNT gives the number of entries, and PCRE_INFO_NAMEENTRYSIZE gives the size of each entry; both of these return an **int** value. The entry size depends on the length of the longest name. PCRE_INFO_NAMETABLE returns a pointer to the first entry of the table (a pointer to **char**). The first two bytes of each entry are the number of the capturing parenthesis, most significant byte first. The rest of the entry is the corresponding name, zero terminated. The names are in alphabetical order. When PCRE_DUPNAMES is set, duplicate names are in order of their parentheses numbers. For example, consider the following pattern (assume PCRE_EXTENDED is set, so white space - including newlines - is ignored):

```
(?<date> (?<year>(\d\d)?\d\d) - (?<month>\d\d) - (?<day>\d\d) )
```

There are four named subpatterns, so the table has four entries, and each entry in the table is eight bytes long. The table is as follows, with non-printing bytes shows in hexadecimal, and undefined bytes shown as ??:

00	01	d	а	t	е	00	??
00	05	d	а	У	00	??	??
00	04	m	0	n	t	h	00
00	02	У	е	а	r	00	??

When writing code to extract data from named subpatterns using the name-tonumber map, remember that the length of the entries is likely to be different for each compiled pattern.

PCRE_INF0_OKPARTIAL

Return 1 if the pattern can be used for partial matching, otherwise 0. The fourth argument should point to an **int** variable. The **pcrepartial** documentation lists the restrictions that apply to patterns when partial matching is used.

PCRE_INFO_OPTIONS

Return a copy of the options with which the pattern was compiled. The fourth argument should point to an **unsigned long int** variable. These option bits are those specified in the call to **pcre_compile()**, modified by any top-level option settings within the pattern itself.

A pattern is automatically anchored by PCRE if all of its top-level alternatives begin with one of the following:

unless PCRE_MULTILINE is set
 always
 always
 if PCRE_DOTALL is set and there are no back references to th

For such patterns, the PCRE_ANCHORED bit is set in the options returned by **pcre_fullinfo()**.

PCRE_INFO_SIZE

Return the size of the compiled pattern, that is, the value that was passed as the argument to **pcre_malloc()** when PCRE was getting memory in which to place the compiled data. The fourth argument should point to a **size_t** variable.

PCRE_INF0_STUDYSIZE

Return the size of the data block pointed to by the *study_data* field in a **pcre_extra** block. That is, it is the value that was passed to **pcre_malloc()** when PCRE was getting memory into which to place the data created by **pcre_study()**. The fourth argument should point to a **size_t** variable.

OBSOLETE INFO FUNCTION

int pcre_info(const pcre *code, int *optptr, int *firstcharptr);

The **pcre_info()** function is now obsolete because its interface is too restrictive to return all the available data about a compiled pattern. New programs should use **pcre_fullinfo()** instead. The yield of **pcre_info()** is the number of capturing subpatterns, or one of the following negative numbers:

PCRE_ERROR_NULL	the	argument <i>code</i> was NULL
PCRE_ERROR_BADMAGIC	the	"magic number" was not found

If the *optptr* argument is not NULL, a copy of the options with which the pattern was compiled is placed in the integer it points to (see PCRE_INFO_OPTIONS above).

If the pattern is not anchored and the *firstcharptr* argument is not NULL, it is used to pass back information about the first character of any matched string (see PCRE_INFO_FIRSTBYTE above).

REFERENCE COUNTS

int pcre_refcount(pcre *code, int adjust);

The **pcre_refcount()** function is used to maintain a reference count in the data block that contains a compiled pattern. It is provided for the benefit of applications that operate in an object-oriented manner, where different parts of the application may be using the same compiled pattern, but you want to free the block when they are all done.

When a pattern is compiled, the reference count field is initialized to zero. It is changed only by calling this function, whose action is to add the *adjust* value (which may be positive or negative) to it. The yield of the function is the new value. However, the value of the count is constrained to lie between 0 and 65535, inclusive. If the new value is outside these limits, it is forced to the appropriate limit value.

Except when it is zero, the reference count is not correctly preserved if a pattern is compiled on one host and then transferred to a host whose byte-order is different. (This seems a highly unlikely scenario.)

MATCHING A PATTERN: THE TRADITIONAL FUNCTION

int pcre_exec(const pcre *code, const pcre_extra *extra, const char *subject, int length, int startoffset, int options, int *ovector, int ovecsize);

The function **pcre_exec()** is called to match a subject string against a compiled pattern, which is passed in the *code* argument. If the pattern has been studied, the result of the study should be passed in the *extra* argument. This function is the main matching facility of the library, and it operates in a Perl-like manner. For specialist use there is also an alternative matching function, which is described <u>below</u> in the section about the **pcre_dfa_exec()** function.

In most applications, the pattern will have been compiled (and optionally studied) in the same process that calls **pcre_exec()**. However, it is possible to save compiled patterns and study data, and then use them later in different processes, possibly even on different hosts. For a discussion about this, see the **pcreprecompile** documentation.

Here is an example of a simple call to **pcre_exec()**:

```
int rc;
int ovector[30];
rc = pcre_exec(
                     /* result of pcre_compile() */
  re,
                     /* we didn't study the pattern */
  NULL,
  "some string", /* the subject string */
                     /* the length of the subject string */
  11,
                    /* start at offset 0 in the subject */
/* default options */
  Θ,
  0,
  ovector, /* vector of integers for substring information
/* number of elements (NOT size in bytes) */
                     /* number of elements (NOT size in bytes) */
  30);
```

Extra data for pcre_exec()

If the *extra* argument is not NULL, it must point to a **pcre_extra** data block. The **pcre_study()** function returns such a block (when it doesn't return NULL), but you can also create one for yourself, and pass additional information in it. The **pcre_extra** block contains the following fields (not necessarily in this order):

```
unsigned long int flags;
void *study_data;
unsigned long int match_limit;
unsigned long int match_limit_recursion;
void *callout_data;
const unsigned char *tables;
```

The *flags* field is a bitmap that specifies which of the other fields are set. The flag bits are:

PCRE_EXTRA_STUDY_DATA PCRE_EXTRA_MATCH_LIMIT PCRE_EXTRA_MATCH_LIMIT_RECURSION PCRE_EXTRA_CALLOUT_DATA PCRE_EXTRA_TABLES

Other flag bits should be set to zero. The *study_data* field is set in the **pcre_extra** block that is returned by **pcre_study()**, together with the appropriate flag bit. You should not set this yourself, but you may add to the block by setting the other fields and their corresponding flag bits.

The *match_limit* field provides a means of preventing PCRE from using up a vast amount of resources when running patterns that are not going to match, but

which have a very large number of possibilities in their search trees. The classic example is the use of nested unlimited repeats.

Internally, PCRE uses a function called **match()** which it calls repeatedly (sometimes recursively). The limit set by *match_limit* is imposed on the number of times this function is called during a match, which has the effect of limiting the amount of backtracking that can take place. For patterns that are not anchored, the count restarts from zero for each position in the subject string.

The default value for the limit can be set when PCRE is built; the default default is 10 million, which handles all but the most extreme cases. You can override the default by suppling **pcre_exec()** with a **pcre_extra** block in which *match_limit* is set, and PCRE_EXTRA_MATCH_LIMIT is set in the *flags* field. If the limit is exceeded, **pcre_exec()** returns PCRE_ERROR_MATCHLIMIT.

The *match_limit_recursion* field is similar to *match_limit*, but instead of limiting the total number of times that **match()** is called, it limits the depth of recursion. The recursion depth is a smaller number than the total number of calls, because not all calls to **match()** are recursive. This limit is of use only if it is set smaller than *match_limit*.

Limiting the recursion depth limits the amount of stack that can be used, or, when PCRE has been compiled to use memory on the heap instead of the stack, the amount of heap memory that can be used.

The default value for *match_limit_recursion* can be set when PCRE is built; the default default is the same value as the default for *match_limit*. You can override the default by suppling **pcre_exec()** with a **pcre_extra** block in which *match_limit_recursion* is set, and PCRE_EXTRA_MATCH_LIMIT_RECURSION is set in the *flags* field. If the limit is exceeded, **pcre_exec()** returns PCRE_ERROR_RECURSIONLIMIT.

The *pcre_callout* field is used in conjunction with the "callout" feature, which is described in the **pcrecallout** documentation.

The *tables* field is used to pass a character tables pointer to **pcre_exec()**; this overrides the value that is stored with the compiled pattern. A non-NULL value is stored with the compiled pattern only if custom tables were supplied to **pcre_compile()** via its *tableptr* argument. If NULL is passed to **pcre_exec()** using this mechanism, it forces PCRE's internal tables to be used. This facility is

helpful when re-using patterns that have been saved after compiling with an external set of tables, because the external tables might be at a different address when **pcre_exec()** is called. See the **pcreprecompile** documentation for a discussion of saving compiled patterns for later use.

Option bits for pcre_exec()

The unused bits of the *options* argument for **pcre_exec()** must be zero. The only bits that may be set are PCRE_ANCHORED, PCRE_NEWLINE_*xxx*, PCRE_NOTBOL, PCRE_NOTEOL, PCRE_NOTEMPTY, PCRE_NO_UTF8_CHECK and PCRE_PARTIAL.

PCRE_ANCHORED

The PCRE_ANCHORED option limits **pcre_exec()** to matching at the first matching position. If a pattern was compiled with PCRE_ANCHORED, or turned out to be anchored by virtue of its contents, it cannot be made unachored at matching time.

PCRE_NEWLINE_CR PCRE_NEWLINE_LF PCRE_NEWLINE_CRLF PCRE_NEWLINE_ANYCRLF PCRE_NEWLINE_ANY

These options override the newline definition that was chosen or defaulted when the pattern was compiled. For details, see the description of **pcre_compile()** above. During matching, the newline choice affects the behaviour of the dot, circumflex, and dollar metacharacters. It may also alter the way the match position is advanced after a match failure for an unanchored pattern. When PCRE_NEWLINE_CRLF, PCRE_NEWLINE_ANYCRLF, or PCRE_NEWLINE_ANY is set, and a match attempt fails when the current position is at a CRLF sequence, the match position is advanced by two characters instead of one, in other words, to after the CRLF.

PCRE_NOTBOL

This option specifies that first character of the subject string is not the beginning of a line, so the circumflex metacharacter should not match before it. Setting this without PCRE_MULTILINE (at compile time) causes circumflex never to match. This option affects only the behaviour of the circumflex metacharacter. It

does not affect A.

PCRE_NOTEOL

This option specifies that the end of the subject string is not the end of a line, so the dollar metacharacter should not match it nor (except in multiline mode) a newline immediately before it. Setting this without PCRE_MULTILINE (at compile time) causes dollar never to match. This option affects only the behaviour of the dollar metacharacter. It does not affect \Z or \z.

PCRE_NOTEMPTY

An empty string is not considered to be a valid match if this option is set. If there are alternatives in the pattern, they are tried. If all the alternatives match the empty string, the entire match fails. For example, if the pattern

a?b?

is applied to a string not beginning with "a" or "b", it matches the empty string at the start of the subject. With PCRE_NOTEMPTY set, this match is not valid, so PCRE searches further into the string for occurrences of "a" or "b".

Perl has no direct equivalent of PCRE_NOTEMPTY, but it does make a special case of a pattern match of the empty string within its **split()** function, and when using the /g modifier. It is possible to emulate Perl's behaviour after matching a null string by first trying the match again at the same offset with PCRE_NOTEMPTY and PCRE_ANCHORED, and then if that fails by advancing the starting offset (see below) and trying an ordinary match again. There is some code that demonstrates how to do this in the *pcredemo.c* sample program.

PCRE_NO_UTF8_CHECK

When PCRE_UTF8 is set at compile time, the validity of the subject as a UTF-8 string is automatically checked when **pcre_exec()** is subsequently called. The value of *startoffset* is also checked to ensure that it points to the start of a UTF-8 character. If an invalid UTF-8 sequence of bytes is found, **pcre_exec()** returns the error PCRE_ERROR_BADUTF8. If *startoffset* contains an invalid value, PCRE_ERROR_BADUTF8_OFFSET is returned.

If you already know that your subject is valid, and you want to skip these checks for performance reasons, you can set the PCRE_NO_UTF8_CHECK option

when calling **pcre_exec()**. You might want to do this for the second and subsequent calls to **pcre_exec()** if you are making repeated calls to find all the matches in a single subject string. However, you should be sure that the value of *startoffset* points to the start of a UTF-8 character. When PCRE_NO_UTF8_CHECK is set, the effect of passing an invalid UTF-8 string as a subject, or a value of *startoffset* that does not point to the start of a UTF-8 character, is undefined. Your program may crash.

PCRE_PARTIAL

This option turns on the partial matching feature. If the subject string fails to match the pattern, but at some point during the matching process the end of the subject was reached (that is, the subject partially matches the pattern and the failure to match occurred only because there were not enough subject characters), **pcre_exec()** returns PCRE_ERROR_PARTIAL instead of PCRE_ERROR_NOMATCH. When PCRE_PARTIAL is used, there are restrictions on what may appear in the pattern. These are discussed in the **pcrepartial** documentation.

The string to be matched by pcre_exec()

The subject string is passed to **pcre_exec()** as a pointer in *subject*, a length in *length*, and a starting byte offset in *startoffset*. In UTF-8 mode, the byte offset must point to the start of a UTF-8 character. Unlike the pattern string, the subject may contain binary zero bytes. When the starting offset is zero, the search for a match starts at the beginning of the subject, and this is by far the most common case.

A non-zero starting offset is useful when searching for another match in the same subject by calling **pcre_exec()** again after a previous success. Setting *startoffset* differs from just passing over a shortened string and setting PCRE_NOTBOL in the case of a pattern that begins with any kind of lookbehind. For example, consider the pattern

\Biss\B

which finds occurrences of "iss" in the middle of words. (\B matches only if the current position in the subject is not a word boundary.) When applied to the string "Mississipi" the first call to **pcre_exec()** finds the first occurrence. If

pcre_exec() is called again with just the remainder of the subject, namely "issipi", it does not match, because \B is always false at the start of the subject, which is deemed to be a word boundary. However, if **pcre_exec()** is passed the entire string again, but with *startoffset* set to 4, it finds the second occurrence of "iss" because it is able to look behind the starting point to discover that it is preceded by a letter.

If a non-zero starting offset is passed when the pattern is anchored, one attempt to match at the given offset is made. This can only succeed if the pattern does not require the match to be at the start of the subject.

How pcre_exec() returns captured substrings

In general, a pattern matches a certain portion of the subject, and in addition, further substrings from the subject may be picked out by parts of the pattern. Following the usage in Jeffrey Friedl's book, this is called "capturing" in what follows, and the phrase "capturing subpattern" is used for a fragment of a pattern that picks out a substring. PCRE supports several other kinds of parenthesized subpattern that do not cause substrings to be captured.

Captured substrings are returned to the caller via a vector of integer offsets whose address is passed in *ovector*. The number of elements in the vector is passed in *ovecsize*, which must be a non-negative number. **Note**: this argument is NOT the size of *ovector* in bytes.

The first two-thirds of the vector is used to pass back captured substrings, each substring using a pair of integers. The remaining third of the vector is used as workspace by **pcre_exec()** while matching capturing subpatterns, and is not available for passing back information. The length passed in *ovecsize* should always be a multiple of three. If it is not, it is rounded down.

When a match is successful, information about captured substrings is returned in pairs of integers, starting at the beginning of *ovector*, and continuing up to two-thirds of its length at the most. The first element of a pair is set to the offset of the first character in a substring, and the second is set to the offset of the first character after the end of a substring. The first pair, *ovector[0]* and *ovector[1]*, identify the portion of the subject string matched by the entire pattern. The next pair is used for the first capturing subpattern, and so on. The value returned by

pcre_exec() is one more than the highest numbered pair that has been set. For example, if two substrings have been captured, the returned value is 3. If there are no capturing subpatterns, the return value from a successful match is 1, indicating that just the first pair of offsets has been set.

If a capturing subpattern is matched repeatedly, it is the last portion of the string that it matched that is returned.

If the vector is too small to hold all the captured substring offsets, it is used as far as possible (up to two-thirds of its length), and the function returns a value of zero. In particular, if the substring offsets are not of interest, **pcre_exec()** may be called with *ovector* passed as NULL and *ovecsize* as zero. However, if the pattern contains back references and the *ovector* is not big enough to remember the related substrings, PCRE has to get additional memory for use during matching. Thus it is usually advisable to supply an *ovector*.

The **pcre_info()** function can be used to find out how many capturing subpatterns there are in a compiled pattern. The smallest size for *ovector* that will allow for *n* captured substrings, in addition to the offsets of the substring matched by the whole pattern, is (n+1)*3.

It is possible for capturing subpattern number n+1 to match some part of the subject when subpattern n has not been used at all. For example, if the string "abc" is matched against the pattern (a|(z))(bc) the return from the function is 4, and subpatterns 1 and 3 are matched, but 2 is not. When this happens, both values in the offset pairs corresponding to unused subpatterns are set to -1.

Offset values that correspond to unused subpatterns at the end of the expression are also set to -1. For example, if the string "abc" is matched against the pattern (abc)(x(yz)?)? subpatterns 2 and 3 are not matched. The return from the function is 2, because the highest used capturing subpattern number is 1. However, you can refer to the offsets for the second and third capturing subpatterns if you wish (assuming the vector is large enough, of course).

Some convenience functions are provided for extracting the captured substrings as separate strings. These are described below.

Error return values from pcre_exec()

If **pcre_exec()** fails, it returns a negative number. The following are defined in the header file:

PCRE_ERROR_NOMATCH (-1)

The subject string did not match the pattern.

PCRE_ERROR_NULL (-2)

Either *code* or *subject* was passed as NULL, or *ovector* was NULL and *ovecsize* was not zero.

PCRE_ERROR_BADOPTION (-3)

An unrecognized bit was set in the *options* argument.

PCRE_ERROR_BADMAGIC (-4)

PCRE stores a 4-byte "magic number" at the start of the compiled code, to catch the case when it is passed a junk pointer and to detect when a pattern that was compiled in an environment of one endianness is run in an environment with the other endianness. This is the error that PCRE gives when the magic number is not present.

```
PCRE_ERROR_UNKNOWN_OPCODE (-5)
```

While running the pattern match, an unknown item was encountered in the compiled pattern. This error could be caused by a bug in PCRE or by overwriting of the compiled pattern.

```
PCRE_ERROR_NOMEMORY (-6)
```

If a pattern contains back references, but the *ovector* that is passed to **pcre_exec()** is not big enough to remember the referenced substrings, PCRE gets a block of memory at the start of matching to use for this purpose. If the call via **pcre_malloc()** fails, this error is given. The memory is automatically freed at the end of matching.

```
PCRE_ERROR_NOSUBSTRING (-7)
```

This error is used by the **pcre_copy_substring()**, **pcre_get_substring()**, and **pcre_get_substring_list()** functions (see below). It is never returned by **pcre_exec()**.

```
PCRE_ERROR_MATCHLIMIT (-8)
```

The backtracking limit, as specified by the *match_limit* field in a **pcre_extra** structure (or defaulted) was reached. See the description above.

PCRE_ERROR_CALLOUT (-9)

This error is never generated by **pcre_exec()** itself. It is provided for use by callout functions that want to yield a distinctive error code. See the **pcrecallout** documentation for details.

PCRE_ERROR_BADUTF8 (-10)

A string that contains an invalid UTF-8 byte sequence was passed as a subject.

PCRE_ERROR_BADUTF8_OFFSET (-11)

The UTF-8 byte sequence that was passed as a subject was valid, but the value of *startoffset* did not point to the beginning of a UTF-8 character.

```
PCRE_ERROR_PARTIAL (-12)
```

The subject string did not match, but it did match partially. See the **<u>pcrepartial</u>** documentation for details of partial matching.

PCRE_ERROR_BADPARTIAL (-13)

The PCRE_PARTIAL option was used with a compiled pattern containing items that are not supported for partial matching. See the **pcrepartial** documentation for details of partial matching.

```
PCRE_ERROR_INTERNAL (-14)
```

An unexpected internal error has occurred. This error could be caused by a bug in PCRE or by overwriting of the compiled pattern.

PCRE_ERROR_BADCOUNT (-15)

This error is given if the value of the *ovecsize* argument is negative.

PCRE_ERROR_RECURSIONLIMIT (-21)

The internal recursion limit, as specified by the *match_limit_recursion* field in a **pcre_extra** structure (or defaulted) was reached. See the description above.

```
PCRE_ERROR_NULLWSLIMIT (-22)
```

When a group that can match an empty substring is repeated with an unbounded

upper limit, the subject position at the start of the group must be remembered, so that a test for an empty string can be made when the end of the group is reached. Some workspace is required for this; if it runs out, this error is given.

PCRE_ERROR_BADNEWLINE (-23)

An invalid combination of PCRE_NEWLINE_*xxx* options was given.

Error numbers -16 to -20 are not used by **pcre_exec()**.

EXTRACTING CAPTURED SUBSTRINGS BY NUMBER

int pcre_copy_substring(const char *subject, int *ovector, int stringcount, int stringnumber, char *buffer, int buffersize);

int pcre_get_substring(const char *subject, int *ovector, int stringcount, int stringnumber, const char **stringptr);

int pcre_get_substring_list(const char *subject, int *ovector, int stringcount, const char ***listptr);

Captured substrings can be accessed directly by using the offsets returned by **pcre_exec()** in *ovector*. For convenience, the functions **pcre_copy_substring()**, **pcre_get_substring()**, and **pcre_get_substring_list()** are provided for extracting captured substrings as new, separate, zero-terminated strings. These functions identify substrings by number. The next section describes functions for extracting named substrings.

A substring that contains a binary zero is correctly extracted and has a further zero added on the end, but the result is not, of course, a C string. However, you can process such a string by referring to the length that is returned by **pcre_copy_substring()** and **pcre_get_substring()**. Unfortunately, the interface to **pcre_get_substring_list()** is not adequate for handling strings containing binary zeros, because the end of the final string is not independently indicated.

The first three arguments are the same for all three of these functions: *subject* is the subject string that has just been successfully matched, *ovector* is a pointer to the vector of integer offsets that was passed to **pcre_exec()**, and *stringcount* is the number of substrings that were captured by the match, including the substring that matched the entire regular expression. This is the value returned

by **pcre_exec()** if it is greater than zero. If **pcre_exec()** returned zero, indicating that it ran out of space in *ovector*, the value passed as *stringcount* should be the number of elements in the vector divided by three.

The functions **pcre_copy_substring()** and **pcre_get_substring()** extract a single substring, whose number is given as *stringnumber*. A value of zero extracts the substring that matched the entire pattern, whereas higher values extract the captured substrings. For **pcre_copy_substring()**, the string is placed in *buffer*, whose length is given by *buffersize*, while for **pcre_get_substring()** a new block of memory is obtained via **pcre_malloc**, and its address is returned via *stringptr*. The yield of the function is the length of the string, not including the terminating zero, or one of these error codes:

PCRE_ERROR_NOMEMORY (-6)

The buffer was too small for **pcre_copy_substring()**, or the attempt to get memory failed for **pcre_get_substring()**.

PCRE_ERROR_NOSUBSTRING (-7)

There is no substring whose number is *stringnumber*.

The **pcre_get_substring_list()** function extracts all available substrings and builds a list of pointers to them. All this is done in a single block of memory that is obtained via **pcre_malloc**. The address of the memory block is returned via *listptr*, which is also the start of the list of string pointers. The end of the list is marked by a NULL pointer. The yield of the function is zero if all went well, or the error code

PCRE_ERROR_NOMEMORY (-6)

if the attempt to get the memory block failed.

When any of these functions encounter a substring that is unset, which can happen when capturing subpattern number n+1 matches some part of the subject, but subpattern n has not been used at all, they return an empty string. This can be distinguished from a genuine zero-length substring by inspecting the appropriate offset in *ovector*, which is negative for unset substrings.

The two convenience functions **pcre_free_substring()** and **pcre_free_substring_list()** can be used to free the memory returned by a

previous call of **pcre_get_substring()** or **pcre_get_substring_list()**, respectively. They do nothing more than call the function pointed to by **pcre_free**, which of course could be called directly from a C program. However, PCRE is used in some situations where it is linked via a special interface to another programming language that cannot use **pcre_free** directly; it is for these cases that the functions are provided.

EXTRACTING CAPTURED SUBSTRINGS BY NAME

int pcre_get_stringnumber(const pcre *code, const char *name);

int pcre_copy_named_substring(const pcre *code, const char *subject, int *ovector, int stringcount, const char *stringname, char *buffer, int buffersize);

int pcre_get_named_substring(const pcre *code, const char *subject, int *ovector, int stringcount, const char *stringname, const char **stringptr);

To extract a substring by name, you first have to find associated number. For example, for this pattern

(a+)b(?<xxx>\d+)...

the number of the subpattern called "xxx" is 2. If the name is known to be unique (PCRE_DUPNAMES was not set), you can find the number from the name by calling **pcre_get_stringnumber()**. The first argument is the compiled pattern, and the second is the name. The yield of the function is the subpattern number, or PCRE_ERROR_NOSUBSTRING (-7) if there is no subpattern of that name.

Given the number, you can extract the substring directly, or use one of the functions described in the previous section. For convenience, there are also two functions that do the whole job.

Most of the arguments of **pcre_copy_named_substring()** and **pcre_get_named_substring()** are the same as those for the similarly named functions that extract by number. As these are described in the previous section, they are not re-described here. There are just two differences:

First, instead of a substring number, a substring name is given. Second, there is

an extra argument, given at the start, which is a pointer to the compiled pattern. This is needed in order to gain access to the name-to-number translation table.

These functions call **pcre_get_stringnumber()**, and if it succeeds, they then call **pcre_copy_substring()** or **pcre_get_substring()**, as appropriate. **NOTE:** If PCRE_DUPNAMES is set and there are duplicate names, the behaviour may not be what you want (see the next section).

DUPLICATE SUBPATTERN NAMES

int pcre_get_stringtable_entries(const pcre *code, const char *name, char **first, char **last);

When a pattern is compiled with the PCRE_DUPNAMES option, names for subpatterns are not required to be unique. Normally, patterns with duplicate names are such that in any one match, only one of the named subpatterns participates. An example is shown in the **pcrepattern** documentation. When duplicates are present, **pcre_copy_named_substring()** and **pcre_get_named_substring()** return the first substring corresponding to the given name that is set. If none are set, an empty string is returned. The **pcre_get_stringnumber()** function returns one of the numbers that are associated with the name, but it is not defined which it is.

If you want to get full details of all captured substrings for a given name, you must use the **pcre_get_stringtable_entries()** function. The first argument is the compiled pattern, and the second is the name. The third and fourth are pointers to variables which are updated by the function. After it has run, they point to the first and last entries in the name-to-number table for the given name. The function itself returns the length of each entry, or

PCRE_ERROR_NOSUBSTRING (-7) if there are none. The format of the table is described above in the section entitled *Information about a pattern*. Given all the relevant entries for the name, you can extract each of their numbers, and hence the captured data, if any.

FINDING ALL POSSIBLE MATCHES

The traditional matching function uses a similar algorithm to Perl, which stops

when it finds the first match, starting at a given point in the subject. If you want to find all possible matches, or the longest possible match, consider using the alternative matching function (see below) instead. If you cannot use the alternative function, but still need to find all possible matches, you can kludge it up by making use of the callout facility, which is described in the **pcrecallout** documentation.

What you have to do is to insert a callout right at the end of the pattern. When your callout function is called, extract and save the current matched substring. Then return 1, which forces **pcre_exec()** to backtrack and try other alternatives. Ultimately, when it runs out of matches, **pcre_exec()** will yield PCRE_ERROR_NOMATCH.

MATCHING A PATTERN: THE ALTERNATIVE FUNCTION

int pcre_dfa_exec(const pcre *code, const pcre_extra *extra, const char *subject, int length, int startoffset, int options, int *ovector, int ovecsize, int *workspace, int wscount);

The function **pcre_dfa_exec()** is called to match a subject string against a compiled pattern, using a matching algorithm that scans the subject string just once, and does not backtrack. This has different characteristics to the normal algorithm, and is not compatible with Perl. Some of the features of PCRE patterns are not supported. Nevertheless, there are times when this kind of matching can be useful. For a discussion of the two matching algorithms, see the **pcrematching** documentation.

The arguments for the **pcre_dfa_exec()** function are the same as for **pcre_exec()**, plus two extras. The *ovector* argument is used in a different way, and this is described below. The other common arguments are used in the same way as for **pcre_exec()**, so their description is not repeated here.

The two additional arguments provide workspace for the function. The workspace vector should contain at least 20 elements. It is used for keeping track of multiple paths through the pattern tree. More workspace will be needed for patterns and subjects where there are a lot of potential matches.

Here is an example of a simple call to **pcre_dfa_exec()**:

Option bits for pcre_dfa_exec()

The unused bits of the *options* argument for **pcre_dfa_exec()** must be zero. The only bits that may be set are PCRE_ANCHORED, PCRE_NEWLINE_*xxx*, PCRE_NOTBOL, PCRE_NOTEOL, PCRE_NOTEMPTY, PCRE_NO_UTF8_CHECK, PCRE_PARTIAL, PCRE_DFA_SHORTEST, and PCRE_DFA_RESTART. All but the last three of these are the same as for **pcre_exec()**, so their description is not repeated here.

PCRE_PARTIAL

This has the same general effect as it does for **pcre_exec()**, but the details are slightly different. When PCRE_PARTIAL is set for **pcre_dfa_exec()**, the return code PCRE_ERROR_NOMATCH is converted into PCRE_ERROR_PARTIAL if the end of the subject is reached, there have been no complete matches, but there is still at least one matching possibility. The portion of the string that provided the partial match is set as the first matching string.

PCRE_DFA_SHORTEST

Setting the PCRE_DFA_SHORTEST option causes the matching algorithm to stop as soon as it has found one match. Because of the way the alternative algorithm works, this is necessarily the shortest possible match at the first possible matching point in the subject string.

```
PCRE_DFA_RESTART
```

When pcre_dfa_exec() is called with the PCRE_PARTIAL option, and returns a

partial match, it is possible to call it again, with additional subject characters, and have it continue with the same match. The PCRE_DFA_RESTART option requests this action; when it is set, the *workspace* and *wscount* options must reference the same vector as before because data about the match so far is left in them after a partial match. There is more discussion of this facility in the **pcrepartial** documentation.

Successful returns from pcre_dfa_exec()

When **pcre_dfa_exec()** succeeds, it may have matched more than one substring in the subject. Note, however, that all the matches from one run of the function start at the same point in the subject. The shorter matches are all initial substrings of the longer matches. For example, if the pattern

<.*>

is matched against the string

This is <something> <something else> <something further> no more

the three matched strings are

<something> <something> <something else> <something> <something else> <something further>

On success, the yield of the function is a number greater than zero, which is the number of matched substrings. The substrings themselves are returned in *ovector*. Each string uses two elements; the first is the offset to the start, and the second is the offset to the end. In fact, all the strings have the same start offset. (Space could have been saved by giving this only once, but it was decided to retain some compatibility with the way **pcre_exec()** returns data, even though the meaning of the strings is different.)

The strings are returned in reverse order of length; that is, the longest matching string is given first. If there were too many matches to fit into *ovector*, the yield of the function is zero, and the vector is filled with the longest matches.

Error returns from pcre_dfa_exec()

The **pcre_dfa_exec()** function returns a negative number when it fails. Many of the errors are the same as for **pcre_exec()**, and these are described <u>above</u>. There are in addition the following errors that are specific to **pcre_dfa_exec()**:

PCRE_ERROR_DFA_UITEM (-16)

This return is given if **pcre_dfa_exec()** encounters an item in the pattern that it does not support, for instance, the use of \C or a back reference.

PCRE_ERROR_DFA_UCOND (-17)

This return is given if **pcre_dfa_exec()** encounters a condition item that uses a back reference for the condition, or a test for recursion in a specific group. These are not supported.

```
PCRE_ERROR_DFA_UMLIMIT (-18)
```

This return is given if **pcre_dfa_exec()** is called with an *extra* block that contains a setting of the *match_limit* field. This is not supported (it is meaningless).

PCRE_ERROR_DFA_WSSIZE (-19)

This return is given if **pcre_dfa_exec()** runs out of space in the *workspace* vector.

```
PCRE_ERROR_DFA_RECURSE (-20)
```

When a recursive subpattern is processed, the matching function calls itself recursively, using private vectors for *ovector* and *workspace*. This error is given if the output vector is not large enough. This should be extremely rare, as a vector of size 1000 is used.

SEE ALSO

```
pcrebuild(3), pcrecallout(3), pcrecpp(3)(3), pcrematching(3), pcrepartial(3),
pcreposix(3), pcreprecompile(3), pcresample(3), pcrestack(3).
```

<u>AUTHOR</u>

Philip Hazel

University Computing Service Cambridge CB2 3QH, England.

<u>REVISION</u>

Last updated: 13 June 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcrebuild man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- <u>PCRE BUILD-TIME OPTIONS</u>
- <u>C++ SUPPORT</u>
- <u>UTF-8 SUPPORT</u>
- <u>UNICODE CHARACTER PROPERTY SUPPORT</u>
- CODE VALUE OF NEWLINE
- BUILDING SHARED AND STATIC LIBRARIES
- POSIX MALLOC USAGE
- HANDLING VERY LARGE PATTERNS
- <u>AVOIDING EXCESSIVE STACK USAGE</u>
- LIMITING PCRE RESOURCE USAGE
- CREATING CHARACTER TABLES AT BUILD TIME
- <u>USING EBCDIC CODE</u>
- <u>SEE ALSO</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

PCRE BUILD-TIME OPTIONS

This document describes the optional features of PCRE that can be selected when the library is compiled. They are all selected, or deselected, by providing options to the **configure** script that is run before the **make** command. The complete list of options for **configure** (which includes the standard ones such as the selection of the installation directory) can be obtained by running

./configure --help

The following sections include descriptions of options whose names begin with - enable or --disable. These settings specify changes to the defaults for the

configure command. Because of the way that **configure** works, --enable and -disable always come in pairs, so the complementary option always exists as well, but as it specifies the default, it is not described.

<u>C++ SUPPORT</u>

By default, the **configure** script will search for a C++ compiler and C++ header files. If it finds them, it automatically builds the C++ wrapper library for PCRE. You can disable this by adding

--disable-cpp

to the **configure** command.

UTF-8 SUPPORT

To build PCRE with support for UTF-8 character strings, add

--enable-utf8

to the **configure** command. Of itself, this does not make PCRE treat strings as UTF-8. As well as compiling PCRE with this option, you also have have to set the PCRE_UTF8 option when you call the **pcre_compile()** function.

UNICODE CHARACTER PROPERTY SUPPORT

UTF-8 support allows PCRE to process character values greater than 255 in the strings that it handles. On its own, however, it does not provide any facilities for accessing the properties of such characters. If you want to be able to use the pattern escapes \P, \p, and \X, which refer to Unicode character properties, you must add

--enable-unicode-properties

to the **configure** command. This implies UTF-8 support, even if you have not explicitly requested it.

Including Unicode property support adds around 30K of tables to the PCRE library. Only the general category properties such as *Lu* and *Nd* are supported.

Details are given in the **<u>pcrepattern</u>** documentation.

CODE VALUE OF NEWLINE

By default, PCRE interprets character 10 (linefeed, LF) as indicating the end of a line. This is the normal newline character on Unix-like systems. You can compile PCRE to use character 13 (carriage return, CR) instead, by adding

--enable-newline-is-cr

to the **configure** command. There is also a --enable-newline-is-lf option, which explicitly specifies linefeed as the newline character.

Alternatively, you can specify that line endings are to be indicated by the two character sequence CRLF. If you want this, add

--enable-newline-is-crlf

to the **configure** command. There is a fourth option, specified by

--enable-newline-is-anycrlf

which causes PCRE to recognize any of the three sequences CR, LF, or CRLF as indicating a line ending. Finally, a fifth option, specified by

```
--enable-newline-is-any
```

causes PCRE to recognize any Unicode newline sequence.

Whatever line ending convention is selected when PCRE is built can be overridden when the library functions are called. At build time it is conventional to use the standard for your operating system.

BUILDING SHARED AND STATIC LIBRARIES

The PCRE building process uses **libtool** to build both shared and static Unix libraries by default. You can suppress one of these by adding one of

```
--disable-shared
--disable-static
```

to the **configure** command, as required.

POSIX MALLOC USAGE

When PCRE is called through the POSIX interface (see the **pcreposix** documentation), additional working storage is required for holding the pointers to capturing substrings, because PCRE requires three integers per substring, whereas the POSIX interface provides only two. If the number of expected substrings is small, the wrapper function uses space on the stack, because this is faster than using **malloc()** for each call. The default threshold above which the stack is no longer used is 10; it can be changed by adding a setting such as

--with-posix-malloc-threshold=20

to the **configure** command.

HANDLING VERY LARGE PATTERNS

Within a compiled pattern, offset values are used to point from one part to another (for example, from an opening parenthesis to an alternation metacharacter). By default, two-byte values are used for these offsets, leading to a maximum size for a compiled pattern of around 64K. This is sufficient to handle all but the most gigantic patterns. Nevertheless, some people do want to process enormous patterns, so it is possible to compile PCRE to use three-byte or four-byte offsets by adding a setting such as

--with-link-size=3

to the **configure** command. The value given must be 2, 3, or 4. Using longer offsets slows down the operation of PCRE because it has to load additional bytes when handling them.

AVOIDING EXCESSIVE STACK USAGE

When matching with the **pcre_exec()** function, PCRE implements backtracking by making recursive calls to an internal function called **match()**. In environments where the size of the stack is limited, this can severely limit PCRE's operation. (The Unix environment does not usually suffer from this

problem, but it may sometimes be necessary to increase the maximum stack size. There is a discussion in the **pcrestack** documentation.) An alternative approach to recursion that uses memory from the heap to remember data, instead of using recursive function calls, has been implemented to work round the problem of limited stack size. If you want to build a version of PCRE that works this way, add

--disable-stack-for-recursion

to the **configure** command. With this configuration, PCRE will use the **pcre_stack_malloc** and **pcre_stack_free** variables to call memory management functions. By default these point to **malloc()** and **free()**, but you can replace the pointers so that your own functions are used.

Separate functions are provided rather than using **pcre_malloc** and **pcre_free** because the usage is very predictable: the block sizes requested are always the same, and the blocks are always freed in reverse order. A calling program might be able to implement optimized functions that perform better than **malloc()** and **free()**. PCRE runs noticeably more slowly when built in this way. This option affects only the **pcre_exec()** function; it is not relevant for the the **pcre_dfa_exec()** function.

LIMITING PCRE RESOURCE USAGE

Internally, PCRE has a function called **match()**, which it calls repeatedly (sometimes recursively) when matching a pattern with the **pcre_exec()** function. By controlling the maximum number of times this function may be called during a single matching operation, a limit can be placed on the resources used by a single call to **pcre_exec()**. The limit can be changed at run time, as described in the **pcreapi** documentation. The default is 10 million, but this can be changed by adding a setting such as

--with-match-limit=500000

to the **configure** command. This setting has no effect on the **pcre_dfa_exec()** matching function.

In some environments it is desirable to limit the depth of recursive calls of **match()** more strictly than the total number of calls, in order to restrict the

maximum amount of stack (or heap, if --disable-stack-for-recursion is specified) that is used. A second limit controls this; it defaults to the value that is set for -- with-match-limit, which imposes no additional constraints. However, you can set a lower limit by adding, for example,

--with-match-limit-recursion=10000

to the **configure** command. This value can also be overridden at run time.

CREATING CHARACTER TABLES AT BUILD TIME

PCRE uses fixed tables for processing characters whose code values are less than 256. By default, PCRE is built with a set of tables that are distributed in the file *pcre_chartables.c.dist*. These tables are for ASCII codes only. If you add

--enable-rebuild-chartables

to the **configure** command, the distributed tables are no longer used. Instead, a program called **dftables** is compiled and run. This outputs the source for new set of tables, created in the default locale of your C runtime system. (This method of replacing the tables does not work if you are cross compiling, because **dftables** is run on the local host. If you need to create alternative tables when cross compiling, you will have to do so "by hand".)

USING EBCDIC CODE

PCRE assumes by default that it will run in an environment where the character code is ASCII (or Unicode, which is a superset of ASCII). PCRE can, however, be compiled to run in an EBCDIC environment by adding

--enable-ebcdic

to the **configure** command. This setting implies --enable-rebuild-chartables.

SEE ALSO

pcreapi(3), pcre_config(3).

AUTHOR

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 05 June 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcre-config man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- <u>SYNOPSIS</u>
- <u>DESCRIPTION</u>
- <u>OPTIONS</u>
- <u>SEE ALSO</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

SYNOPSIS

pcre-config [--prefix] [--exec-prefix] [--version] [--libs] [--libs-posix] [-cflags] [--cflags-posix]

DESCRIPTION

pcre-config returns the configuration of the installed PCRE libraries and the options required to compile a program to use them.

OPTIONS

--**prefix** Writes the directory prefix used in the PCRE installation for architecture independent files (*/usr* on many systems, */usr/local* on some systems) to the standard output.

--**exec-prefix** Writes the directory prefix used in the PCRE installation for architecture dependent files (normally the same as --**prefix**) to the standard output.

--version Writes the version number of the installed PCRE libraries to the standard output.

--libs Writes to the standard output the command line options required to link with PCRE (-lpcre on many systems).

--**libs-posix** Writes to the standard output the command line options required to link with the PCRE posix emulation library (-**lpcreposix -lpcre** on many systems).

--cflags Writes to the standard output the command line options required to compile files that use PCRE (this may include some -I options, but is blank on many systems).

--cflags-posix Writes to the standard output the command line options required to compile files that use the PCRE posix emulation library (this may include some -I options, but is blank on many systems).

SEE ALSO

pcre(3)

<u>AUTHOR</u>

This manual page was originally written by Mark Baker for the Debian GNU/Linux system. It has been slightly revised as a generic PCRE man page.

REVISION

Last updated: 18 April 2007

Return to the <u>PCRE index page</u>.

pcrecallout man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- <u>PCRE CALLOUTS</u>
- MISSING CALLOUTS
- THE CALLOUT INTERFACE
- <u>RETURN VALUES</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

PCRE CALLOUTS

int (*pcre_callout)(pcre_callout_block *);

PCRE provides a feature called "callout", which is a means of temporarily passing control to the caller of PCRE in the middle of pattern matching. The caller of PCRE provides an external function by putting its entry point in the global variable *pcre_callout*. By default, this variable contains NULL, which disables all calling out.

Within a regular expression, (?C) indicates the points at which the external function is to be called. Different callout points can be identified by putting a number less than 256 after the letter C. The default value is zero. For example, this pattern has two callout points:

(?C1)abc(?C2)def

If the PCRE_AUTO_CALLOUT option bit is set when **pcre_compile()** is called, PCRE automatically inserts callouts, all with number 255, before each item in the pattern. For example, if PCRE_AUTO_CALLOUT is used with the pattern

A(\d{2}|--)

it is processed as if it were

(?C255)A(?C255)((?C255)\d{2}(?C255)-(?C255)-(?C255))(?C255)

Notice that there is a callout before and after each parenthesis and alternation bar. Automatic callouts can be used for tracking the progress of pattern matching. The **pcretest** command has an option that sets automatic callouts; when it is used, the output indicates how the pattern is matched. This is useful information when you are trying to optimize the performance of a particular pattern.

MISSING CALLOUTS

You should be aware that, because of optimizations in the way PCRE matches patterns, callouts sometimes do not happen. For example, if the pattern is

ab(?C4)cd

PCRE knows that any matching string must contain the letter "d". If the subject string is "abyz", the lack of "d" means that matching doesn't ever start, and the callout is never reached. However, with "abyd", though the result is still no match, the callout is obeyed.

THE CALLOUT INTERFACE

During matching, when PCRE reaches a callout point, the external function defined by *pcre_callout* is called (if it is set). This applies to both the **pcre_exec()** and the **pcre_dfa_exec()** matching functions. The only argument to the callout function is a pointer to a **pcre_callout** block. This structure contains the following fields:

int	version;
int	callout_number;
int	*offset_vector;
const char	*subject;
int	<pre>subject_length;</pre>
int	start_match;
int	current_position;

int	capture_top;
int	capture_last;
void	<pre>*callout_data;</pre>
int	<pre>pattern_position;</pre>
int	<pre>next_item_length;</pre>

The *version* field is an integer containing the version number of the block format. The initial version was 0; the current version is 1. The version number will change again in future if additional fields are added, but the intention is never to remove any of the existing fields.

The *callout_number* field contains the number of the callout, as compiled into the pattern (that is, the number after ?C for manual callouts, and 255 for automatically generated callouts).

The *offset_vector* field is a pointer to the vector of offsets that was passed by the caller to **pcre_exec()** or **pcre_dfa_exec()**. When **pcre_exec()** is used, the contents can be inspected in order to extract substrings that have been matched so far, in the same way as for extracting substrings after a match has completed. For **pcre_dfa_exec()** this field is not useful.

The *subject* and *subject_length* fields contain copies of the values that were passed to **pcre_exec()**.

The *start_match* field normally contains the offset within the subject at which the current match attempt started. However, if the escape sequence \K has been encountered, this value is changed to reflect the modified starting point. If the pattern is not anchored, the callout function may be called several times from the same point in the pattern for different starting points in the subject.

The *current_position* field contains the offset within the subject of the current match pointer.

When the **pcre_exec()** function is used, the *capture_top* field contains one more than the number of the highest numbered captured substring so far. If no substrings have been captured, the value of *capture_top* is one. This is always the case when **pcre_dfa_exec()** is used, because it does not support captured substrings.

The *capture_last* field contains the number of the most recently captured substring. If no substrings have been captured, its value is -1. This is always the

case when **pcre_dfa_exec()** is used.

The *callout_data* field contains a value that is passed to **pcre_exec()** or **pcre_dfa_exec()** specifically so that it can be passed back in callouts. It is passed in the *pcre_callout* field of the **pcre_extra** data structure. If no such data was passed, the value of *callout_data* in a **pcre_callout** block is NULL. There is a description of the **pcre_extra** structure in the **pcreapi** documentation.

The *pattern_position* field is present from version 1 of the *pcre_callout* structure. It contains the offset to the next item to be matched in the pattern string.

The *next_item_length* field is present from version 1 of the *pcre_callout* structure. It contains the length of the next item to be matched in the pattern string. When the callout immediately precedes an alternation bar, a closing parenthesis, or the end of the pattern, the length is zero. When the callout precedes an opening parenthesis, the length is that of the entire subpattern.

The *pattern_position* and *next_item_length* fields are intended to help in distinguishing between different automatic callouts, which all have the same callout number. However, they are set for all callouts.

RETURN VALUES

The external callout function returns an integer to PCRE. If the value is zero, matching proceeds as normal. If the value is greater than zero, matching fails at the current point, but the testing of other matching possibilities goes ahead, just as if a lookahead assertion had failed. If the value is less than zero, the match is abandoned, and **pcre_exec()** (or **pcre_dfa_exec()**) returns the negative value.

Negative values should normally be chosen from the set of PCRE_ERROR_xxx values. In particular, PCRE_ERROR_NOMATCH forces a standard "no match" failure. The error number PCRE_ERROR_CALLOUT is reserved for use by callout functions; it will never be used by PCRE itself.

<u>AUTHOR</u>

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 29 May 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcrecompat man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

DIFFERENCES BETWEEN PCRE AND PERL

This document describes the differences in the ways that PCRE and Perl handle regular expressions. The differences described here are mainly with respect to Perl 5.8, though PCRE versions 7.0 and later contain some features that are expected to be in the forthcoming Perl 5.10.

1. PCRE has only a subset of Perl's UTF-8 and Unicode support. Details of what it does have are given in the <u>section on UTF-8 support</u> in the main <u>pcre</u> page.

2. PCRE does not allow repeat quantifiers on lookahead assertions. Perl permits them, but they do not mean what you might think. For example, (?!a){3} does not assert that the next three characters are not "a". It just asserts that the next character is not "a" three times.

3. Capturing subpatterns that occur inside negative lookahead assertions are counted, but their entries in the offsets vector are never set. Perl sets its numerical variables from any such patterns that are matched before the assertion fails to match something (thereby succeeding), but only if the negative lookahead assertion contains just one branch.

4. Though binary zero characters are supported in the subject string, they are not allowed in a pattern string because it is passed as a normal C string, terminated by zero. The escape sequence \0 can be used in the pattern to represent a binary zero.

5. The following Perl escape sequences are not supported: \l, \u, \L, \U, and \N. In fact these are implemented by Perl's general string-handling and are not part of its pattern matching engine. If any of these are encountered by PCRE, an error

is generated.

6. The Perl escape sequences \p, \P, and \X are supported only if PCRE is built with Unicode character property support. The properties that can be tested with \p and \P are limited to the general category properties such as Lu and Nd, script names such as Greek or Han, and the derived properties Any and L&.

7. PCRE does support the \Q...\E escape for quoting substrings. Characters in between are treated as literals. This is slightly different from Perl in that \$ and @ are also handled as literals inside the quotes. In Perl, they cause variable interpolation (but of course PCRE does not have variables). Note the following examples:

Pattern	PCRE matches	Perl matches
\Qabc\$xyz\E	abc\$xyz	abc followed by the content
\Qabc\\$xyz\E	abc\\$xyz	abc\\$xyz
\Qabc\E\\$\Qxyz\E	abc\$xyz	abc\$xyz

The \Q...\E sequence is recognized both inside and outside character classes.

8. Fairly obviously, PCRE does not support the (?{code}) and (??{code}) constructions. However, there is support for recursive patterns. This is not available in Perl 5.8, but will be in Perl 5.10. Also, the PCRE "callout" feature allows an external function to be called during pattern matching. See the **pcrecallout** documentation for details.

9. Subpatterns that are called recursively or as "subroutines" are always treated as atomic groups in PCRE. This is like Python, but unlike Perl.

10. There are some differences that are concerned with the settings of captured strings when part of a pattern is repeated. For example, matching "aba" against the pattern $/^(a(b)?)+$ in Perl leaves \$2 unset, but in PCRE it is set to "b".

11. PCRE provides some extensions to the Perl regular expression facilities. Perl 5.10 will include new features that are not in earlier versions, some of which (such as named parentheses) have been in PCRE for some time. This list is with respect to Perl 5.10:

(a) Although lookbehind assertions must match fixed length strings, each alternative branch of a lookbehind assertion can match a different length of

string. Perl requires them all to have the same length.

(b) If PCRE_DOLLAR_ENDONLY is set and PCRE_MULTILINE is not set, the \$ meta-character matches only at the very end of the string.

(c) If PCRE_EXTRA is set, a backslash followed by a letter with no special meaning is faulted. Otherwise, like Perl, the backslash is quietly ignored. (Perl can be made to issue a warning.)

(d) If PCRE_UNGREEDY is set, the greediness of the repetition quantifiers is inverted, that is, by default they are not greedy, but if followed by a question mark they are.

(e) PCRE_ANCHORED can be used at matching time to force a pattern to be tried only at the first matching position in the subject string.

(f) The PCRE_NOTBOL, PCRE_NOTEOL, PCRE_NOTEMPTY, and PCRE_NO_AUTO_CAPTURE options for **pcre_exec()** have no Perl equivalents.

(g) The callout facility is PCRE-specific.

(h) The partial matching facility is PCRE-specific.

(i) Patterns compiled by PCRE can be saved and re-used at a later time, even on different hosts that have the other endianness.

(j) The alternative matching function (**pcre_dfa_exec()**) matches in a different way and is not Perl-compatible.

AUTHOR

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 13 June 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcrecpp man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- <u>SYNOPSIS OF C++ WRAPPER</u>
- DESCRIPTION
- <u>MATCHING INTERFACE</u>
- **QUOTING METACHARACTERS**
- PARTIAL MATCHES
- <u>UTF-8 AND THE MATCHING INTERFACE</u>
- <u>PASSING MODIFIERS TO THE REGULAR EXPRESSION ENGINE</u>
- SCANNING TEXT INCREMENTALLY
- PARSING HEX/OCTAL/C-RADIX NUMBERS
- <u>REPLACING PARTS OF STRINGS</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

SYNOPSIS OF C++ WRAPPER

#include <pcrecpp.h>

DESCRIPTION

The C++ wrapper for PCRE was provided by Google Inc. Some additional functionality was added by Giuseppe Maxia. This brief man page was constructed from the notes in the *pcrecpp.h* file, which should be consulted for further details.

MATCHING INTERFACE

The "FullMatch" operation checks that supplied text matches a supplied pattern exactly. If pointer arguments are supplied, it copies matched sub-strings that match sub-patterns into them.

```
Example: successful match
  pcrecpp::RE re("h.*o");
  re.FullMatch("hello");
Example: unsuccessful match (requires full match):
  pcrecpp::RE re("e");
  !re.FullMatch("hello");
Example: creating a temporary RE object:
  pcrecpp::RE("h.*o").FullMatch("hello");
```

You can pass in a "const char*" or a "string" for "text". The examples below tend to use a const char*. You can, as in the different examples above, store the RE object explicitly in a variable or use a temporary RE object. The examples below use one mode or the other arbitrarily. Either could correctly be used for any of these examples.

You must supply extra pointer arguments to extract matched subpieces.

```
Example: extracts "ruby" into "s" and 1234 into "i"
    int i;
    string s;
    pcrecpp::RE re("(\\w+):(\\d+)");
    re.FullMatch("ruby:1234", &s;, &i;);
Example: does not try to extract any extra sub-patterns
    re.FullMatch("ruby:1234", &s;);
Example: does not try to extract into NULL
    re.FullMatch("ruby:1234", NULL, &i;);
Example: integer overflow causes failure
    !re.FullMatch("ruby:1234567891234", NULL, &i;);
Example: fails because there aren't enough sub-patterns:
    !pcrecpp::RE("\\w+:\\d+").FullMatch("ruby:1234", &s;);
Example: fails because string cannot be stored in integer
    !pcrecpp::RE("(.*)").FullMatch("ruby", &i;);
```

The provided pointer arguments can be pointers to any scalar numeric type, or one of:

string	(matched piece is copied to string)
StringPiece	(StringPiece is mutated to point to matched piece)
Т	(where "bool T::ParseFrom(const char*, int)" exists
NULL	(the corresponding matched sub-pattern is not copie

The function returns true iff all of the following conditions are satisfied:

- a. "text" matches "pattern" exactly;
- b. The number of matched sub-patterns is >= number of supplied pointers;
- c. The "i"th argument has a suitable type for holding the string captured as the "i"th sub-pattern. If you pass in NULL for the "i"th argument, or pass fewer arguments than number of sub-patterns, "i"th captured sub-pattern is ignored.

CAVEAT: An optional sub-pattern that does not exist in the matched string is assigned the empty string. Therefore, the following will return false (because the empty string is not a valid number):

```
int number;
pcrecpp::RE::FullMatch("abc", "[a-z]+(\\d+)?", &number;);
```

The matching interface supports at most 16 arguments per call. If you need more, consider using the more general interface **pcrecpp::RE::DoMatch**. See **pcrecpp.h** for the signature for **DoMatch**.

QUOTING METACHARACTERS

You can use the "QuoteMeta" operation to insert backslashes before all potentially meaningful characters in a string. The returned string, used as a regular expression, will exactly match the original string.

```
Example:
    string quoted = RE::QuoteMeta(unquoted);
```

Note that it's legal to escape a character even if it has no special meaning in a regular expression -- so this function does that. (This also makes it identical to the perl function of the same name; see "perldoc -f quotemeta".) For example, "1.5-2.0?" becomes "1.5-2.0?".

PARTIAL MATCHES

You can use the "PartialMatch" operation when you want the pattern to match any substring of the text.

```
Example: simple search for a string:
    pcrecpp::RE("ell").PartialMatch("hello");
Example: find first number in a string:
    int number;
    pcrecpp::RE re("(\\d+)");
    re.PartialMatch("x*100 + 20", &number;);
    assert(number == 100);
```

UTF-8 AND THE MATCHING INTERFACE

By default, pattern and text are plain text, one byte per character. The UTF8 flag, passed to the constructor, causes both pattern and string to be treated as UTF-8 text, still a byte stream but potentially multiple bytes per character. In practice, the text is likelier to be UTF-8 than the pattern, but the match returned may depend on the UTF8 flag, so always use it when matching UTF8 text. For example, "." will match one byte normally but with UTF8 set may match up to three bytes of a multi-byte character.

```
Example:
    pcrecpp::RE_Options options;
    options.set_utf8();
    pcrecpp::RE re(utf8_pattern, options);
    re.FullMatch(utf8_string);
Example: using the convenience function UTF8():
    pcrecpp::RE re(utf8_pattern, pcrecpp::UTF8());
    re.FullMatch(utf8_string);
```

NOTE: The UTF8 flag is ignored if pcre was not configured with the

--enable-utf8 flag.

PASSING MODIFIERS TO THE REGULAR EXPRESSION ENGINE

PCRE defines some modifiers to change the behavior of the regular expression engine. The C++ wrapper defines an auxiliary class, RE_Options, as a vehicle to

pass such modifiers to a RE class. Currently, the following modifiers are supported:

modifier	description	Perl correspondin
PCRE_CASELESS PCRE_MULTILINE PCRE_DOTALL PCRE_DOLLAR_ENDONLY PCRE_EXTRA PCRE_EXTENDED PCRE_UTF8 PCRE_UNGREEDY PCRE_NO_AUTO_CAPTURE	<pre>case insensitive match multiple lines match dot matches newlines \$ matches only at end strict escape parsing ignore whitespaces handles UTF8 chars reverses * and *? disables capturing parens</pre>	/i /m /s N/A N/A /x built-in N/A N/A (*)

(*) Both Perl and PCRE allow non capturing parentheses by means of the "?:" modifier within the pattern itself. e.g. (?:ab|cd) does not capture, while (ab|cd) does.

For a full account on how each modifier works, please check the PCRE API reference page.

For each modifier, there are two member functions whose name is made out of the modifier in lowercase, without the "PCRE_" prefix. For instance, PCRE_CASELESS is handled by

```
bool caseless()
```

which returns true if the modifier is set, and

```
RE_Options & set_caseless(bool)
```

which sets or unsets the modifier. Moreover, PCRE_EXTRA_MATCH_LIMIT can be accessed through the **set_match_limit()** and **match_limit()** member functions. Setting *match_limit* to a non-zero value will limit the execution of pcre to keep it from doing bad things like blowing the stack or taking an eternity to return a result. A value of 5000 is good enough to stop stack blowup in a 2MB thread stack. Setting *match_limit* to zero disables match limiting. Alternatively, you can call **match_limit_recursion()** which uses

PCRE_EXTRA_MATCH_LIMIT_RECURSION to limit how much PCRE recurses. **match_limit()** limits the number of matches PCRE does; **match_limit_recursion()** limits the depth of internal recursion, and therefore the amount of stack that is used.

Normally, to pass one or more modifiers to a RE class, you declare a *RE_Options* object, set the appropriate options, and pass this object to a RE constructor. Example:

```
RE_options opt;
opt.set_caseless(true);
if (RE("HELLO", opt).PartialMatch("hello world")) ...
```

RE_options has two constructors. The default constructor takes no arguments and creates a set of flags that are off by default. The optional parameter *option_flags* is to facilitate transfer of legacy code from C programs. This lets you do

```
RE(pattern,
    RE_Options(PCRE_CASELESS|PCRE_MULTILINE)).PartialMatch(str);
```

However, new code is better off doing

```
RE(pattern,
    RE_Options().set_caseless(true).set_multiline(true))
    .PartialMatch(str);
```

If you are going to pass one of the most used modifiers, there are some convenience functions that return a RE_Options class with the appropriate modifier already set: **CASELESS()**, **UTF8()**, **MULTILINE()**, **DOTALL()**, and **EXTENDED()**.

If you need to set several options at once, and you don't want to go through the pains of declaring a RE_Options object and setting several options, there is a parallel method that give you such ability on the fly. You can concatenate several **set_xxxxx()** member functions, since each of them returns a reference to its class object. For example, to pass PCRE_CASELESS, PCRE_EXTENDED, and PCRE_MULTILINE to a RE with one statement, you may write:

```
RE(" ^ xyz \\s+ .* blah$",
RE_Options()
   .set_caseless(true)
   .set_extended(true)
   .set_multiline(true)).PartialMatch(sometext);
```

SCANNING TEXT INCREMENTALLY

The "Consume" operation may be useful if you want to repeatedly match regular expressions at the front of a string and skip over them as they match. This requires use of the "StringPiece" type, which represents a sub-range of a real string. Like RE, StringPiece is defined in the pcrecpp namespace.

Each successful call to "Consume" will set "var/value", and also advance "input" so it points past the matched text.

The "FindAndConsume" operation is similar to "Consume" but does not anchor your match at the beginning of the string. For example, you could extract all words from a string by repeatedly calling

```
pcrecpp::RE("(\\w+)").FindAndConsume(&input;, &word;)
```

PARSING HEX/OCTAL/C-RADIX NUMBERS

By default, if you pass a pointer to a numeric value, the corresponding text is interpreted as a base-10 number. You can instead wrap the pointer with a call to one of the operators Hex(), Octal(), or CRadix() to interpret the text in another base. The CRadix operator interprets C-style "0" (base-8) and "0x" (base-16) prefixes, but defaults to base-10.

will leave 64 in a, b, c, and d.

REPLACING PARTS OF STRINGS

You can replace the first match of "pattern" in "str" with "rewrite". Within "rewrite", backslash-escaped digits (\1 to \9) can be used to insert text matching corresponding parenthesized group from the pattern. \0 in "rewrite" refers to the entire matching text. For example:

```
string s = "yabba dabba doo";
pcrecpp::RE("b+").Replace("d", &s;);
```

will leave "s" containing "yada dabba doo". The result is true if the pattern matches and a replacement occurs, false otherwise.

GlobalReplace is like **Replace** except that it replaces all occurrences of the pattern in the string with the rewrite. Replacements are not subject to rematching. For example:

```
string s = "yabba dabba doo";
pcrecpp::RE("b+").GlobalReplace("d", &s;);
```

will leave "s" containing "yada dada doo". It returns the number of replacements made.

Extract is like **Replace**, except that if the pattern matches, "rewrite" is copied into "out" (an additional argument) with substitutions. The non-matching portions of "text" are ignored. Returns true iff a match occurred and the extraction happened successfully; if no match occurs, the string is left unaffected.

<u>AUTHOR</u>

The C++ wrapper was contributed by Google Inc. Copyright © 2007 Google Inc.

REVISION

Last updated: 06 March 2007

Return to the <u>PCRE index page</u>.

pcregrep man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- <u>SYNOPSIS</u>
- <u>DESCRIPTION</u>
- <u>OPTIONS</u>
- ENVIRONMENT VARIABLES
- <u>NEWLINES</u>
- **OPTIONS COMPATIBILITY**
- OPTIONS WITH DATA
- MATCHING ERRORS
- <u>DIAGNOSTICS</u>
- <u>SEE ALSO</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

<u>SYNOPSIS</u>

pcregrep [options] [long options] [pattern] [path1 path2 ...]

DESCRIPTION

pcregrep searches files for character patterns, in the same way as other grep commands do, but it uses the PCRE regular expression library to support patterns that are compatible with the regular expressions of Perl 5. See **pcrepattern(3)** for a full description of syntax and semantics of the regular expressions that PCRE supports.

Patterns, whether supplied on the command line or in a separate file, are given without delimiters. For example:

pcregrep Thursday /etc/motd

If you attempt to use delimiters (for example, by surrounding a pattern with slashes, as is common in Perl scripts), they are interpreted as part of the pattern. Quotes can of course be used on the command line because they are interpreted by the shell, and indeed they are required if a pattern contains white space or shell metacharacters.

The first argument that follows any option settings is treated as the single pattern to be matched when neither **-e** nor **-f** is present. Conversely, when one or both of these options are used to specify patterns, all arguments are treated as path names. At least one of **-e**, **-f**, or an argument pattern must be provided.

If no files are specified, **pcregrep** reads the standard input. The standard input can also be referenced by a name consisting of a single hyphen. For example:

pcregrep some-pattern /file1 - /file3

By default, each line that matches the pattern is copied to the standard output, and if there is more than one file, the file name is output at the start of each line. However, there are options that can change how **pcregrep** behaves. In particular, the **-M** option makes it possible to search for patterns that span line boundaries. What defines a line boundary is controlled by the **-N** (**--newline**) option.

Patterns are limited to 8K or BUFSIZ characters, whichever is the greater. BUFSIZ is defined in **<stdio.h**>.

If the **LC_ALL** or **LC_CTYPE** environment variable is set, **pcregrep** uses the value to set a locale when calling the PCRE library. The **--locale** option can be used to override this.

OPTIONS

-- This terminate the list of options. It is useful if the next item on the command line starts with a hyphen but is not an option. This allows for the processing of patterns and filenames that start with hyphens.

-A *number*, --after-context=*number* Output *number* lines of context after each matching line. If filenames and/or line numbers are being output, a hyphen separator is used instead of a colon for the context lines. A line containing "--" is

output between each group of lines, unless they are in fact contiguous in the input file. The value of *number* is expected to be relatively small. However, **pcregrep** guarantees to have up to 8K of following text available for context output.

-**B** number, --**before-context**=number Output number lines of context before each matching line. If filenames and/or line numbers are being output, a hyphen separator is used instead of a colon for the context lines. A line containing "--" is output between each group of lines, unless they are in fact contiguous in the input file. The value of *number* is expected to be relatively small. However, **pcregrep** guarantees to have up to 8K of preceding text available for context output.

-C *number*, **--context**=*number* Output *number* lines of context both before and after each matching line. This is equivalent to setting both **-A** and **-B** to the same value.

-c, **--count** Do not output individual lines; instead just output a count of the number of lines that would otherwise have been output. If several files are given, a count is output for each of them. In this mode, the **-A**, **-B**, and **-C** options are ignored.

--**colour**, --**color** If this option is given without any data, it is equivalent to "-- colour=auto". If data is required, it must be given in the same shell item, separated by an equals sign.

--colour=value, --color=value This option specifies under what circumstances the part of a line that matched a pattern should be coloured in the output. The value may be "never" (the default), "always", or "auto". In the latter case, colouring happens only if the standard output is connected to a terminal. The colour can be specified by setting the environment variable PCREGREP_COLOUR or PCREGREP_COLOR. The value of this variable should be a string of two numbers, separated by a semicolon. They are copied directly into the control string for setting colour on a terminal, so it is your responsibility to ensure that they make sense. If neither of the environment variables is set, the default is "1;31", which gives red.

-D *action*, **--devices**=*action* If an input path is not a regular file or a directory, "action" specifies how it is to be processed. Valid values are "read" (the default)

or "skip" (silently skip the path).

-d action, --directories=action If an input path is a directory, "action" specifies how it is to be processed. Valid values are "read" (the default), "recurse" (equivalent to the -r option), or "skip" (silently skip the path). In the default case, directories are read as if they were ordinary files. In some operating systems the effect of reading a directory like this is an immediate end-of-file.

-e pattern, --regex=pattern, --regexp=pattern Specify a pattern to be matched. This option can be used multiple times in order to specify several patterns. It can also be used as a way of specifying a single pattern that starts with a hyphen. When -e is used, no argument pattern is taken from the command line; all arguments are treated as file names. There is an overall maximum of 100 patterns. They are applied to each line in the order in which they are defined until one matches (or fails to match if -v is used). If -f is used with -e, the command line patterns are matched first, followed by the patterns from the file, independent of the order in which these options are specified. Note that multiple use of -e is not the same as a single pattern with alternatives. For example, X|Y finds the first character in a line that is X or Y, whereas if the two patterns are given separately, pcregrep finds X if it is present, even if it follows Y in the line. It finds Y only if there is no X in the line. This really matters only if you are using -o to show the portion of the line that matched.

--**exclude**=*pattern* When **pcregrep** is searching the files in a directory as a consequence of the **-r** (recursive search) option, any files whose names match the pattern are excluded. The pattern is a PCRE regular expression. If a file name matches both --**include** and --**exclude**, it is excluded. There is no short form for this option.

-F, **--fixed-strings** Interpret each pattern as a list of fixed strings, separated by newlines, instead of as a regular expression. The **-w** (match as a word) and **-x** (match whole line) options can be used with **-F**. They apply to each of the fixed strings. A line is selected if any of the fixed strings are found in it (subject to **-w** or **-x**, if present).

-f *filename*, **--file**=*filename* Read a number of patterns from the file, one per line, and match them against each line of input. A data line is output if any of the patterns match it. The filename can be given as "-" to refer to the standard input. When **-f** is used, patterns specified on the command line using **-e** may also be

present; they are tested before the file's patterns. However, no other pattern is taken from the command line; all arguments are treated as file names. There is an overall maximum of 100 patterns. Trailing white space is removed from each line, and blank lines are ignored. An empty file contains no patterns and therefore matches nothing.

-H, **--with-filename** Force the inclusion of the filename at the start of output lines when searching a single file. By default, the filename is not shown in this case. For matching lines, the filename is followed by a colon and a space; for context lines, a hyphen separator is used. If a line number is also being output, it follows the file name without a space.

-h, **--no-filename** Suppress the output filenames when searching multiple files. By default, filenames are shown when multiple files are searched. For matching lines, the filename is followed by a colon and a space; for context lines, a hyphen separator is used. If a line number is also being output, it follows the file name without a space.

--help Output a brief help message and exit.

-i, --ignore-case Ignore upper/lower case distinctions during comparisons.

--include=*pattern* When **pcregrep** is searching the files in a directory as a consequence of the **-r** (recursive search) option, only those files whose names match the pattern are included. The pattern is a PCRE regular expression. If a file name matches both --include and --exclude, it is excluded. There is no short form for this option.

-L, **--files-without-match** Instead of outputting lines from the files, just output the names of the files that do not contain any lines that would have been output. Each file name is output once, on a separate line.

-l, **--files-with-matches** Instead of outputting lines from the files, just output the names of the files containing lines that would have been output. Each file name is output once, on a separate line. Searching stops as soon as a matching line is found in a file.

--label=name This option supplies a name to be used for the standard input when file names are being output. If not supplied, "(standard input)" is used. There is no short form for this option.

--locale=locale-name This option specifies a locale to be used for pattern matching. It overrides the value in the LC_ALL or LC_CTYPE environment variables. If no locale is specified, the PCRE library's default (usually the "C" locale) is used. There is no short form for this option.

-M, --multiline Allow patterns to match more than one line. When this option is given, patterns may usefully contain literal newline characters and internal occurrences of ^ and \$ characters. The output for any one match may consist of more than one line. When this option is set, the PCRE library is called in "multiline" mode. There is a limit to the number of lines that can be matched, imposed by the way that **pcregrep** buffers the input file as it scans it. However, **pcregrep** ensures that at least 8K characters or the rest of the document (whichever is the shorter) are available for forward matching, and similarly the previous 8K characters (or all the previous characters, if fewer than 8K) are guaranteed to be available for lookbehind assertions.

-**N** *newline-type*, --**newline**=*newline-type* The PCRE library supports five different conventions for indicating the ends of lines. They are the single-character sequences CR (carriage return) and LF (linefeed), the two-character sequence CRLF, an "anycrlf" convention, which recognizes any of the preceding three types, and an "any" convention, in which any Unicode line ending sequence is assumed to end a line. The Unicode sequences are the three just mentioned, plus VT (vertical tab, U+000B), FF (formfeed, U+000C), NEL (next line, U+0085), LS (line separator, U+2028), and PS (paragraph separator, U+2029).

When the PCRE library is built, a default line-ending sequence is specified. This is normally the standard sequence for the operating system. Unless otherwise specified by this option, **pcregrep** uses the library's default. The possible values for this option are CR, LF, CRLF, ANYCRLF, or ANY. This makes it possible to use **pcregrep** on files that have come from other environments without having to modify their line endings. If the data that is being scanned does not agree with the convention set by this option, **pcregrep** may behave in strange ways.

-n, **--line-number** Precede each output line by its line number in the file, followed by a colon and a space for matching lines or a hyphen and a space for context lines. If the filename is also being output, it precedes the line number.

-o, --only-matching Show only the part of the line that matched a pattern. In this

mode, no context is shown. That is, the **-A**, **-B**, and **-C** options are ignored.

-q, **--quiet** Work quietly, that is, display nothing except error messages. The exit status indicates whether or not any matches were found.

-r, **--recursive** If any given path is a directory, recursively scan the files it contains, taking note of any **--include** and **--exclude** settings. By default, a directory is read as a normal file; in some operating systems this gives an immediate end-of-file. This option is a shorthand for setting the **-d** option to "recurse".

-s, **--no-messages** Suppress error messages about non-existent or unreadable files. Such files are quietly skipped. However, the return code is still 2, even if matches were found in other files.

-u, **--utf-8** Operate in UTF-8 mode. This option is available only if PCRE has been compiled with UTF-8 support. Both patterns and subject lines must be valid strings of UTF-8 characters.

-V, **--version** Write the version numbers of **pcregrep** and the PCRE library that is being used to the standard error stream.

-v, **--invert-match** Invert the sense of the match, so that lines which do *not* match any of the patterns are the ones that are found.

-w, **--word-regex**, **--word-regexp** Force the patterns to match only whole words. This is equivalent to having \b at the start and end of the pattern.

-x, **--line-regex**, **--line-regexp** Force the patterns to be anchored (each must start matching at the beginning of a line) and in addition, require them to match entire lines. This is equivalent to having ^ and \$ characters at the start and end of each alternative branch in every pattern.

ENVIRONMENT VARIABLES

The environment variables **LC_ALL** and **LC_CTYPE** are examined, in that order, for a locale. The first one that is set is used. This can be overridden by the **--locale** option. If no locale is set, the PCRE library's default (usually the "C" locale) is used.

NEWLINES

The **-N** (--**newline**) option allows **pcregrep** to scan files with different newline conventions from the default. However, the setting of this option does not affect the way in which **pcregrep** writes information to the standard error and output streams. It uses the string "\n" in C **printf()** calls to indicate newlines, relying on the C I/O library to convert this to an appropriate sequence if the output is sent to a file.

OPTIONS COMPATIBILITY

The majority of short and long forms of **pcregrep**'s options are the same as in the GNU **grep** program. Any long option of the form **--xxx-regexp** (GNU terminology) is also available as **--xxx-regex** (PCRE terminology). However, the **--locale**, **-M**, **--multiline**, **-u**, and **--utf-8** options are specific to **pcregrep**.

OPTIONS WITH DATA

There are four different ways in which an option with data can be specified. If a short form option is used, the data may follow immediately, or in the next command line item. For example:

```
-f/some/file
-f /some/file
```

If a long form option is used, the data may appear in the same command line item, separated by an equals character, or (with one exception) it may appear in the next command line item. For example:

```
--file=/some/file
--file /some/file
```

Note, however, that if you want to supply a file name beginning with \sim as data in a shell command, and have the shell expand \sim to a home directory, you must separate the file name from the option, because the shell does not treat \sim specially unless it is at the start of an item.

The exception to the above is the --colour (or --color) option, for which the data

is optional. If this option does have data, it must be given in the first form, using an equals character. Otherwise it will be assumed that it has no data.

MATCHING ERRORS

It is possible to supply a regular expression that takes a very long time to fail to match certain lines. Such patterns normally involve nested indefinite repeats, for example: (a+)*\d when matched against a line of a's with no final digit. The PCRE matching function has a resource limit that causes it to abort in these circumstances. If this happens, **pcregrep** outputs an error message and the line that caused the problem to the standard error stream. If there are more than 20 such errors, **pcregrep** gives up.

DIAGNOSTICS

Exit status is 0 if any matches were found, 1 if no matches were found, and 2 for syntax errors and non-existent or inacessible files (even if matches were found in other files) or too many matching errors. Using the **-s** option to suppress error messages about inaccessible files does not affect the return code.

SEE ALSO

pcrepattern(3), pcretest(1).

<u>AUTHOR</u>

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 16 April 2007 Copyright © 1997-2007 University of Cambridge. Return to the <u>PCRE index page</u>.

pcrematching man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- <u>PCRE MATCHING ALGORITHMS</u>
- <u>REGULAR EXPRESSIONS AS TREES</u>
- THE STANDARD MATCHING ALGORITHM
- THE ALTERNATIVE MATCHING ALGORITHM
- <u>ADVANTAGES OF THE ALTERNATIVE ALGORITHM</u>
- DISADVANTAGES OF THE ALTERNATIVE ALGORITHM
- <u>AUTHOR</u>
- <u>REVISION</u>

PCRE MATCHING ALGORITHMS

This document describes the two different algorithms that are available in PCRE for matching a compiled regular expression against a given subject string. The "standard" algorithm is the one provided by the **pcre_exec()** function. This works in the same was as Perl's matching function, and provides a Perl-compatible matching operation.

An alternative algorithm is provided by the **pcre_dfa_exec()** function; this operates in a different way, and is not Perl-compatible. It has advantages and disadvantages compared with the standard algorithm, and these are described below.

When there is only one possible way in which a given subject string can match a pattern, the two algorithms give the same answer. A difference arises, however, when there are multiple possibilities. For example, if the pattern

^<.*>

is matched against the string

<something> <something else> <something further>

there are three possible answers. The standard algorithm finds only one of them, whereas the alternative algorithm finds all three.

REGULAR EXPRESSIONS AS TREES

The set of strings that are matched by a regular expression can be represented as a tree structure. An unlimited repetition in the pattern makes the tree of infinite size, but it is still a tree. Matching the pattern to a given subject string (from a given starting point) can be thought of as a search of the tree. There are two ways to search a tree: depth-first and breadth-first, and these correspond to the two matching algorithms provided by PCRE.

THE STANDARD MATCHING ALGORITHM

In the terminology of Jeffrey Friedl's book "Mastering Regular Expressions", the standard algorithm is an "NFA algorithm". It conducts a depth-first search of the pattern tree. That is, it proceeds along a single path through the tree, checking that the subject matches what is required. When there is a mismatch, the algorithm tries any alternatives at the current point, and if they all fail, it backs up to the previous branch point in the tree, and tries the next alternative branch at that level. This often involves backing up (moving to the left) in the subject string as well. The order in which repetition branches are tried is controlled by the greedy or ungreedy nature of the quantifier.

If a leaf node is reached, a matching string has been found, and at that point the algorithm stops. Thus, if there is more than one possible match, this algorithm returns the first one that it finds. Whether this is the shortest, the longest, or some intermediate length depends on the way the greedy and ungreedy repetition quantifiers are specified in the pattern.

Because it ends up with a single path through the tree, it is relatively straightforward for this algorithm to keep track of the substrings that are matched by portions of the pattern in parentheses. This provides support for capturing parentheses and back references.

THE ALTERNATIVE MATCHING ALGORITHM

This algorithm conducts a breadth-first search of the tree. Starting from the first matching point in the subject, it scans the subject string from left to right, once, character by character, and as it does this, it remembers all the paths through the tree that represent valid matches. In Friedl's terminology, this is a kind of "DFA algorithm", though it is not implemented as a traditional finite state machine (it keeps multiple states active simultaneously).

The scan continues until either the end of the subject is reached, or there are no more unterminated paths. At this point, terminated paths represent the different matching possibilities (if there are none, the match has failed). Thus, if there is more than one possible match, this algorithm finds all of them, and in particular, it finds the longest. In PCRE, there is an option to stop the algorithm after the first match (which is necessarily the shortest) has been found.

Note that all the matches that are found start at the same point in the subject. If the pattern

cat(er(pillar)?)

is matched against the string "the caterpillar catchment", the result will be the three strings "cat", "cater", and "caterpillar" that start at the fourth character of the subject. The algorithm does not automatically move on to find matches that start at later positions.

There are a number of features of PCRE regular expressions that are not supported by the alternative matching algorithm. They are as follows:

1. Because the algorithm finds all possible matches, the greedy or ungreedy nature of repetition quantifiers is not relevant. Greedy and ungreedy quantifiers are treated in exactly the same way. However, possessive quantifiers can make a difference when what follows could also match what is quantified, for example in a pattern like this:

^a++\w!

This pattern matches "aaab!" but not "aaa!", which would be matched by a nonpossessive quantifier. Similarly, if an atomic group is present, it is matched as if it were a standalone pattern at the current point, and the longest match is then "locked in" for the rest of the overall pattern. 2. When dealing with multiple paths through the tree simultaneously, it is not straightforward to keep track of captured substrings for the different matching possibilities, and PCRE's implementation of this algorithm does not attempt to do this. This means that no captured substrings are available.

3. Because no substrings are captured, back references within the pattern are not supported, and cause errors if encountered.

4. For the same reason, conditional expressions that use a backreference as the condition or test for a specific group recursion are not supported.

5. Because many paths through the tree may be active, the \K escape sequence, which resets the start of the match when encountered (but may be on some paths and not on others), is not supported. It causes an error if encountered.

6. Callouts are supported, but the value of the *capture_top* field is always 1, and the value of the *capture_last* field is always -1.

7. The \C escape sequence, which (in the standard algorithm) matches a single byte, even in UTF-8 mode, is not supported because the alternative algorithm moves through the subject string one character at a time, for all active paths through the tree.

ADVANTAGES OF THE ALTERNATIVE ALGORITHM

Using the alternative matching algorithm provides the following advantages:

1. All possible matches (at a single point in the subject) are automatically found, and in particular, the longest match is found. To find more than one match using the standard algorithm, you have to do kludgy things with callouts.

2. There is much better support for partial matching. The restrictions on the content of the pattern that apply when using the standard algorithm for partial matching do not apply to the alternative algorithm. For non-anchored patterns, the starting position of a partial match is available.

3. Because the alternative algorithm scans the subject string just once, and never needs to backtrack, it is possible to pass very long subject strings to the matching function in several pieces, checking for partial matching each time.

DISADVANTAGES OF THE ALTERNATIVE ALGORITHM

The alternative algorithm suffers from a number of disadvantages:

1. It is substantially slower than the standard algorithm. This is partly because it has to search for all possible matches, but is also because it is less susceptible to optimization.

2. Capturing parentheses and back references are not supported.

3. Although atomic groups are supported, their use does not provide the performance advantage that it does for the standard algorithm.

<u>AUTHOR</u>

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 29 May 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcrepartial man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- PARTIAL MATCHING IN PCRE
- <u>RESTRICTED PATTERNS FOR PCRE_PARTIAL</u>
- EXAMPLE OF PARTIAL MATCHING USING PCRETEST
- <u>MULTI-SEGMENT MATCHING WITH pcre_dfa_exec()</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

PARTIAL MATCHING IN PCRE

In normal use of PCRE, if the subject string that is passed to **pcre_exec()** or **pcre_dfa_exec()** matches as far as it goes, but is too short to match the entire pattern, PCRE_ERROR_NOMATCH is returned. There are circumstances where it might be helpful to distinguish this case from other cases in which there is no match.

Consider, for example, an application where a human is required to type in data for a field with specific formatting requirements. An example might be a date in the form *ddmmmyy*, defined by this pattern:

^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d\$

If the application sees the user's keystrokes one by one, and can check that what has been typed so far is potentially valid, it is able to raise an error as soon as a mistake is made, possibly beeping and not reflecting the character that has been typed. This immediate feedback is likely to be a better user interface than a check that is delayed until the entire string has been entered.

PCRE supports the concept of partial matching by means of the PCRE_PARTIAL option, which can be set when calling **pcre_exec()** or

pcre_dfa_exec(). When this flag is set for **pcre_exec()**, the return code PCRE_ERROR_NOMATCH is converted into PCRE_ERROR_PARTIAL if at any time during the matching process the last part of the subject string matched part of the pattern. Unfortunately, for non-anchored matching, it is not possible to obtain the position of the start of the partial match. No captured data is set when PCRE_ERROR_PARTIAL is returned.

When PCRE_PARTIAL is set for **pcre_dfa_exec()**, the return code PCRE_ERROR_NOMATCH is converted into PCRE_ERROR_PARTIAL if the end of the subject is reached, there have been no complete matches, but there is still at least one matching possibility. The portion of the string that provided the partial match is set as the first matching string.

Using PCRE_PARTIAL disables one of PCRE's optimizations. PCRE remembers the last literal byte in a pattern, and abandons matching immediately if such a byte is not present in the subject string. This optimization cannot be used for a subject string that might match only partially.

RESTRICTED PATTERNS FOR PCRE_PARTIAL

Because of the way certain internal optimizations are implemented in the **pcre_exec()** function, the PCRE_PARTIAL option cannot be used with all patterns. These restrictions do not apply when **pcre_dfa_exec()** is used. For **pcre_exec()**, repeated single characters such as

a{2,4}

and repeated single metasequences such as

\d+

are not permitted if the maximum number of occurrences is greater than one. Optional items such as \d? (where the maximum is one) are permitted. Quantifiers with any values are permitted after parentheses, so the invalid examples above can be coded thus:

(a){2,4} (\d)+

These constructions run more slowly, but for the kinds of application that are envisaged for this facility, this is not felt to be a major restriction. If PCRE_PARTIAL is set for a pattern that does not conform to the restrictions, **pcre_exec()** returns the error code PCRE_ERROR_BADPARTIAL (-13). You can use the PCRE_INFO_OKPARTIAL call to **pcre_fullinfo()** to find out if a compiled pattern can be used for partial matching.

EXAMPLE OF PARTIAL MATCHING USING PCRETEST

If the escape sequence \P is present in a **pcretest** data line, the PCRE_PARTIAL flag is used for the match. Here is a run of **pcretest** that uses the date example quoted above:

```
re> /^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d
data> 25jun04\P
0: 25jun04
1: jun
data> 25dec3\P
Partial match
data> 3ju\P
Partial match
data> 3juj\P
No match
data> j\P
No match
```

The first data string is matched completely, so **pcretest** shows the matched substrings. The remaining four strings do not match the complete pattern, but the first two are partial matches. The same test, using **pcre_dfa_exec()** matching (by means of the \D escape sequence), produces the following output:

```
re> /^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d
data> 25jun04\P\D
0: 25jun04
data> 23dec3\P\D
Partial match: 23dec3
data> 3ju\P\D
Partial match: 3ju
data> 3juj\P\D
No match
data> j\P\D
No match
```

Notice that in this case the portion of the string that was matched is made available.

MULTI-SEGMENT MATCHING WITH pcre_dfa_exec()

When a partial match has been found using **pcre_dfa_exec()**, it is possible to continue the match by providing additional subject data and calling **pcre_dfa_exec()** again with the same compiled regular expression, this time setting the PCRE_DFA_RESTART option. You must also pass the same working space as before, because this is where details of the previous partial match are stored. Here is an example using **pcretest**, using the \R escape sequence to set the PCRE_DFA_RESTART option (\P and \D are as above):

```
re> /^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d
data> 23ja\P\D
Partial match: 23ja
data> n05\R\D
0: n05
```

The first call has "23ja" as the subject, and requests partial matching; the second call has "n05" as the subject for the continued (restarted) match. Notice that when the match is complete, only the last part is shown; PCRE does not retain the previously partially-matched string. It is up to the calling program to do that if it needs to.

You can set PCRE_PARTIAL with PCRE_DFA_RESTART to continue partial matching over multiple segments. This facility can be used to pass very long subject strings to **pcre_dfa_exec()**. However, some care is needed for certain types of pattern.

1. If the pattern contains tests for the beginning or end of a line, you need to pass the PCRE_NOTBOL or PCRE_NOTEOL options, as appropriate, when the subject string for any call does not contain the beginning or end of a line.

2. If the pattern contains backward assertions (including \b or \B), you need to arrange for some overlap in the subject strings to allow for this. For example, you could pass the subject in chunks that are 500 bytes long, but in a buffer of 700 bytes, with the starting offset set to 200 and the previous 200 bytes at the start of the buffer.

3. Matching a subject string that is split into multiple segments does not always produce exactly the same result as matching over one single long string. The difference arises when there are multiple matching possibilities, because a partial

match result is given only when there are no completed matches in a call to **pcre_dfa_exec()**. This means that as soon as the shortest match has been found, continuation to a new subject segment is no longer possible. Consider this **pcretest** example:

```
re> /dog(sbody)?/
data> do\P\D
Partial match: do
data> gsb\R\P\D
0: g
data> dogsbody\D
0: dogsbody
1: dog
```

The pattern matches the words "dog" or "dogsbody". When the subject is presented in several parts ("do" and "gsb" being the first two) the match stops when "dog" has been found, and it is not possible to continue. On the other hand, if "dogsbody" is presented as a single string, both matches are found.

Because of this phenomenon, it does not usually make sense to end a pattern that is going to be matched in this way with a variable repeat.

4. Patterns that contain alternatives at the top level which do not all start with the same pattern item may not work as expected. For example, consider this pattern:

1234|3789

If the first part of the subject is "ABC123", a partial match of the first alternative is found at offset 3. There is no partial match for the second alternative, because such a match does not start at the same point in the subject string. Attempting to continue with the string "789" does not yield a match because only those alternatives that match at one point in the subject are remembered. The problem arises because the start of the second alternative matches within the first alternative. There is no problem with anchored patterns or patterns such as:

```
1234 | ABCD
```

where no string can be a partial match for both alternatives.

<u>AUTHOR</u>

Philip Hazel

University Computing Service Cambridge CB2 3QH, England.

<u>REVISION</u>

Last updated: 04 June 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcrepattern man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- <u>PCRE REGULAR EXPRESSION DETAILS</u>
- CHARACTERS AND METACHARACTERS
- <u>BACKSLASH</u>
- CIRCUMFLEX AND DOLLAR
- FULL STOP (PERIOD, DOT)
- MATCHING A SINGLE BYTE
- SQUARE BRACKETS AND CHARACTER CLASSES
- POSIX CHARACTER CLASSES
- <u>VERTICAL BAR</u>
- INTERNAL OPTION SETTING
- <u>SUBPATTERNS</u>
- DUPLICATE SUBPATTERN NUMBERS
- <u>NAMED SUBPATTERNS</u>
- <u>REPETITION</u>
- <u>ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS</u>
- BACK REFERENCES
- ASSERTIONS
- <u>CONDITIONAL SUBPATTERNS</u>
- <u>COMMENTS</u>
- <u>RECURSIVE PATTERNS</u>
- SUBPATTERNS AS SUBROUTINES
- <u>CALLOUTS</u>
- <u>SEE ALSO</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

PCRE REGULAR EXPRESSION DETAILS

The syntax and semantics of the regular expressions supported by PCRE are described below. Regular expressions are also described in the Perl documentation and in a number of books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly, covers regular expressions in great detail. This description of PCRE's regular expressions is intended as reference material.

The original operation of PCRE was on strings of one-byte characters. However, there is now also support for UTF-8 character strings. To use this, you must build PCRE to include UTF-8 support, and then call **pcre_compile()** with the PCRE_UTF8 option. How this affects pattern matching is mentioned in several places below. There is also a summary of UTF-8 features in the <u>section on UTF-8 support</u> in the main **pcre** page.

The remainder of this document discusses the patterns that are supported by PCRE when its main matching function, **pcre_exec()**, is used. From release 6.0, PCRE offers a second matching function, **pcre_dfa_exec()**, which matches using a different algorithm that is not Perl-compatible. Some of the features discussed below are not available when **pcre_dfa_exec()** is used. The advantages and disadvantages of the alternative function, and how it differs from the normal function, are discussed in the **pcrematching** page.

CHARACTERS AND METACHARACTERS

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

The quick brown fox

matches a portion of a subject string that is identical to itself. When caseless matching is specified (the PCRE_CASELESS option), letters are matched independently of case. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *metacharacters*, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized within square brackets. Outside square brackets, the metacharacters are as follows:

\ \$ [()	<pre>general escape character with several uses assert start of string (or line, in multiline mode) assert end of string (or line, in multiline mode) match any character except newline (by default) start character class definition start of alternative branch start subpattern and subpattern</pre>
	end subpattern extends the meaning of (
?	also 0 or 1 quantifier also quantifier minimizer
*	0 or more quantifier
+	1 or more quantifier also "possessive quantifier"
{	start min/max quantifier

Part of a pattern that is in square brackets is called a "character class". In a character class the only metacharacters are:

\mathbf{X}	general escape character
٨	negate the class, but only if the first character
-	indicates character range
[POSIX character class (only if followed by POSIX syntax)
]	terminates the character class

The following sections describe the use of each of the metacharacters.

BACKSLASH

The backslash character has several uses. Firstly, if it is followed by a nonalphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes. For example, if you want to match a * character, you write * in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a non-alphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write \\.

If a pattern is compiled with the PCRE_EXTENDED option, whitespace in the pattern (other than in a character class) and characters between a # outside a character class and the next newline are ignored. An escaping backslash can be used to include a whitespace or # character as part of the pattern.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. This is different from Perl in that \$ and @ are handled as literals in \Q...\E sequences in PCRE, whereas in Perl, \$ and @ cause variable interpolation. Note the following examples:

Pattern	PCRE matches	Perl matches
\Qabc\$xyz\E	abc\$xyz	abc followed by the contents of
\Qabc\\$xyz\E	abc\\$xyz	abc\\$xyz
\Qabc\E\\$\Qxyz\E	abc\$xyz	abc\$xyz

The \Q...\E sequence is recognized both inside and outside character classes.

Non-printing characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

```
alarm, that is, the BEL character (hex 07)
\a
          "control-x", where x is any character
\cx
          escape (hex 1B)
\e
\f
          formfeed (hex OC)
          newline (hex OA)
n 
          carriage return (hex OD)
\r
          tab (hex 09)
\t
          character with octal code ddd, or backreference
\ddd
          character with hex code hh
\lambda xhh
\x{hhh..} character with hex code hhh..
```

The precise effect of cx is as follows: if x is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus cz becomes hex 1A, but c becomes hex 3B, while c; becomes hex 7B.

After x, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between $x{ and }$, but the value of the character code must be less than 256 in non-UTF-8 mode, and less than 2**31 in UTF-8 mode (that is, the maximum hexadecimal value is 7FFFFFF). If characters other than hexadecimal digits appear between $x{ and }$, or if there is no terminating }, this form of escape is not recognized. Instead, the initial x will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

Characters whose value is less than 256 can be defined by either of the two syntaxes for x. There is no difference in the way they are handled. For example, xdc is exactly the same as $x{dc}$.

After \0 up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence \0\x\07 specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is given <u>later</u>, following the discussion of <u>parenthesized subpatterns</u>.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and uses them to generate a data character. Any subsequent digits stand for themselves. In non-UTF-8 mode, the value of a character specified in octal must be less than \400. In UTF-8 mode, values up to \777 are permitted. For example:

\040	is another way of writing a space
\40	is the same, provided there are fewer than 40 previous capt
\7	is always a back reference

\11	might be a back reference, or another way of writing a tab
\011	is always a tab
\0113	is a tab followed by the character "3"
\113	might be a back reference, otherwise the character with oct
\377	might be a back reference, otherwise the byte consisting en
\81	is either a back reference, or a binary zero followed by th

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single character value can be used both inside and outside character classes. In addition, inside a character class, the sequence \b is interpreted as the backspace character (hex 08), and the sequences \R and \X are interpreted as the characters "R" and "X", respectively. Outside a character class, these sequences have different meanings (see below).

Absolute and relative back references

The sequence \g followed by a positive or negative number, optionally enclosed in braces, is an absolute or relative back reference. A named back reference can be coded as \g{name}. Back references are discussed <u>later</u>, following the discussion of <u>parenthesized subpatterns</u>.

Generic character types

Another use of backslash is for specifying generic character types. The following are always recognized:

\d	any decimal digit
\D	any character that is not a decimal digit
\h	any horizontal whitespace character
\H	any character that is not a horizontal whitespace character
\s	any whitespace character
\S	any character that is not a whitespace character
\v	any vertical whitespace character
\V	any character that is not a vertical whitespace character
\w	any "word" character
\W	any "non-word" character

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

For compatibility with Perl, \s does not match the VT character (code 11). This makes it different from the the POSIX "space" class. The \s characters are HT (9), LF (10), FF (12), CR (13), and space (32). If "use locale;" is included in a Perl script, \s may match the VT character. In PCRE, it never does.

In UTF-8 mode, characters with values greater than 128 never match \d, \s, or \w, and always match \D, \S, and \W. This is true even when Unicode character property support is available. These sequences retain their original meanings from before UTF-8 support was available, mainly for efficiency reasons.

The sequences \h, \H, \v, and \V are Perl 5.10 features. In contrast to the other sequences, these do match certain high-valued codepoints in UTF-8 mode. The horizontal space characters are:

U+0009 U+0020	Horizontal tab Space
U+00A0	Non-break space
U+1680	Ogham space mark
U+180E	Mongolian vowel separator
U+2000	En quad
U+2001	Em quad
U+2002	En space
U+2003	Em space
U+2004	Three-per-em space
U+2005	Four-per-em space
U+2006	Six-per-em space
U+2007	Figure space
U+2008	Punctuation space
U+2009	Thin space
U+200A	Hair space
U+202F	Narrow no-break space
U+205F	Medium mathematical space
U+3000	Ideographic space

The vertical space characters are:

U+000A	Linefeed	
U+000B	Vertical	tab
U+000C	Formfeed	
U+000D	Carriage	return

U+0085	Next line
U+2028	Line separator
U+2029	Paragraph separator

A "word" character is an underscore or any character less than 256 that is a letter or digit. The definition of letters and digits is controlled by PCRE's low-valued character tables, and may vary if locale-specific matching is taking place (see "Locale support" in the **pcreapi** page). For example, in a French locale such as "fr_FR" in Unix-like systems, or "french" in Windows, some character codes greater than 128 are used for accented letters, and these are matched by \w. The use of locales with Unicode is discouraged.

Newline sequences

Outside a character class, the escape sequence \R matches any Unicode newline sequence. This is a Perl 5.10 feature. In non-UTF-8 mode \R is equivalent to the following:

 $(?>\n|\n|\x0b|\f|\r|\x85)$

This is an example of an "atomic group", details of which are given <u>below</u>. This particular group matches either the two-character sequence CR followed by LF, or one of the single characters LF (linefeed, U+000A), VT (vertical tab, U+000B), FF (formfeed, U+000C), CR (carriage return, U+000D), or NEL (next line, U+0085). The two-character sequence is treated as a single unit that cannot be split.

In UTF-8 mode, two additional characters whose codepoints are greater than 255 are added: LS (line separator, U+2028) and PS (paragraph separator, U+2029). Unicode character property support is not needed for these characters to be recognized.

Inside a character class, \R matches the letter "R".

Unicode character properties

When PCRE is built with Unicode character property support, three additional escape sequences that match characters with specific properties are available. When not in UTF-8 mode, these sequences are of course limited to testing

characters whose codepoints are less than 256, but they do work in this mode. The extra escape sequences are:

\p{ <i>xx</i> }	a character with the xx property
$P{xx}$	a character without the xx property
\X	an extended Unicode sequence

The property names represented by *xx* above are limited to the Unicode script names, the general category properties, and "Any", which matches any character (including newline). Other properties such as "InMusicalSymbols" are not currently supported by PCRE. Note that \P{Any} does not match any characters, so always causes a match failure.

Sets of Unicode characters are defined as belonging to certain scripts. A character from one of these sets can be matched using a script name. For example:

\p{Greek} \P{Han}

Those that are not part of an identified script are lumped together as "Common". The current list of scripts is:

Arabic, Armenian, Balinese, Bengali, Bopomofo, Braille, Buginese, Buhid, Canadian_Aboriginal, Cherokee, Common, Coptic, Cuneiform, Cypriot, Cyrillic, Deseret, Devanagari, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hiragana, Inherited, Kannada, Katakana, Kharoshthi, Khmer, Lao, Latin, Limbu, Linear_B, Malayalam, Mongolian, Myanmar, New_Tai_Lue, Nko, Ogham, Old_Italic, Old_Persian, Oriya, Osmanya, Phags_Pa, Phoenician, Runic, Shavian, Sinhala, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Yi.

Each character has exactly one general category property, specified by a twoletter abbreviation. For compatibility with Perl, negation can be specified by including a circumflex between the opening brace and the property name. For example, p^{Lu} is the same as P^{Lu} .

If only one letter is specified with \p or \P, it includes all the general category properties that start with that letter. In this case, in the absence of negation, the curly brackets in the escape sequence are optional; these two examples have the

same effect:

\p{L} \pL

The following general category property codes are supported:

C	Other
CC	Control
Cf	Format
Cn	Unassigned
Co	Private use
Cs	Surrogate
L	Letter
L1	Lower case letter
LM	Modifier letter
LO	Other letter
Lt	Title case letter
Lu	Upper case letter
M	Mark
Mc	Spacing mark
Me	Enclosing mark
Mn	Non-spacing mark
N	Number
Nd	Decimal number
Nl	Letter number
No	Other number
P Pd Pe Pf Pi Po Ps	Punctuation Connector punctuation Dash punctuation Close punctuation Final punctuation Initial punctuation Other punctuation Open punctuation
S	Symbol
SC	Currency symbol
Sk	Modifier symbol
Sm	Mathematical symbol
SO	Other symbol
Z	Separator
Zl	Line separator
Zp	Paragraph separator

Zs Space separator

The special property L& is also supported: it matches a character that has the Lu, Ll, or Lt property, in other words, a letter that is not classified as a modifier or "other".

The long synonyms for these properties that Perl supports (such as \p{Letter}) are not supported by PCRE, nor is it permitted to prefix any of these properties with "Is".

No character that is in the Unicode table has the Cn (unassigned) property. Instead, this property is assumed for any code point that is not in the Unicode table.

Specifying caseless matching does not affect these escape sequences. For example, \p{Lu} always matches only upper case letters.

The X escape matches any number of Unicode characters that form an extended Unicode sequence. X is equivalent to

(?>\PM\pM*)

That is, it matches a character without the "mark" property, followed by zero or more characters with the "mark" property, and treats the sequence as an atomic group (see below). Characters with the "mark" property are typically accents that affect the preceding character. None of them have codepoints less than 256, so in non-UTF-8 mode \X matches any one character.

Matching characters by Unicode property is not fast, because PCRE has to search a structure that contains data for over fifteen thousand characters. That is why the traditional escape sequences such as \d and \w do not use Unicode properties in PCRE.

Resetting the match start

The escape sequence \K, which is a Perl 5.10 feature, causes any previously matched characters not to be included in the final matched sequence. For example, the pattern:

foo\Kbar

matches "foobar", but reports that it has matched "bar". This feature is similar to a lookbehind assertion (described below). However, in this case, the part of the subject before the real match does not have to be of fixed length, as lookbehind assertions do. The use of \K does not interfere with the setting of <u>captured</u> <u>substrings</u>. For example, when the pattern

(foo)\Kbar

matches "foobar", the first substring is still set to "foo".

Simple assertions

The final use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described <u>below</u>. The backslashed assertions are:

\b	matches at a word boundary
∖B	matches when not at a word boundary
\A	matches at the start of the subject
\Ζ	matches at the end of the subject
	also matches before a newline at the end of the subject
\z	matches only at the end of the subject
\G	matches at the first matching position in the subject

These assertions may not appear in character classes (but note that \b has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match w or W (i.e. one matches w and the other matches W), or the start or end of the string if the first or last character matches w, respectively.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described in the next section) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode. These three assertions are not affected by the PCRE_NOTBOL or PCRE_NOTEOL options, which affect only the behaviour of the circumflex and dollar metacharacters. However, if the *startoffset* argument of **pcre_exec()** is non-zero, indicating that matching is to start at a point other than the beginning of the subject, \A can never match. The difference between \Z and \z is that \Z

matches before a newline at the end of the string as well as at the very end, whereas z matches only at the end.

The \G assertion is true only when the current matching position is at the start point of the match, as specified by the *startoffset* argument of **pcre_exec()**. It differs from \A when the value of *startoffset* is non-zero. By calling **pcre_exec()** multiple times with appropriate arguments, you can mimic Perl's /g option, and it is in this kind of implementation where \G can be useful.

Note, however, that PCRE's interpretation of \G, as the start of the current match, is subtly different from Perl's, which defines it as the end of the previous match. In Perl, these can be different when the previously matched string was empty. Because PCRE does just one match at a time, it cannot reproduce this behaviour.

If all the alternatives of a pattern begin with \G, the expression is anchored to the starting match position, and the "anchored" flag is set in the compiled regular expression.

CIRCUMFLEX AND DOLLAR

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the subject string. If the *startoffset* argument of **pcre_exec()** is non-zero, circumflex can never match if the PCRE_MULTILINE option is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion that is true only if the current matching point is at the end of the subject string, or immediately before a newline at the end of the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class. The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the PCRE_DOLLAR_ENDONLY option at compile time. This does not affect the Z assertion.

The meanings of the circumflex and dollar characters are changed if the PCRE_MULTILINE option is set. When this is the case, a circumflex matches immediately after internal newlines as well as at the start of the subject string. It does not match after a newline that ends the string. A dollar matches before any newlines in the string, as well as at the very end, when PCRE_MULTILINE is set. When newline is specified as the two-character sequence CRLF, isolated CR and LF characters do not indicate newlines.

For example, the pattern //abc\$/ matches the subject string "def/nabc" (where /n represents a newline) in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with ^ are not anchored in multiline mode, and a match for circumflex is possible when the *startoffset* argument of **pcre_exec()** is non-zero. The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set.

Note that the sequences A, Z, and z can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with A it is always anchored, whether or not PCRE_MULTILINE is set.

FULL STOP (PERIOD, DOT)

Outside a character class, a dot in the pattern matches any one character in the subject string except (by default) a character that signifies the end of a line. In UTF-8 mode, the matched character may be more than one byte long.

When a line ending is defined as a single character, dot never matches that character; when the two-character sequence CRLF is used, dot does not match CR if it is immediately followed by LF, but otherwise it matches all characters (including isolated CRs and LFs). When any Unicode line endings are being recognized, dot does not match CR or LF or any of the other line ending characters.

The behaviour of dot with regard to newlines can be changed. If the PCRE_DOTALL option is set, a dot matches any one character, without exception. If the two-character sequence CRLF is present in the subject string, it

takes two dots to match it.

The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newlines. Dot has no special meaning in a character class.

MATCHING A SINGLE BYTE

Outside a character class, the escape sequence \C matches any one byte, both in and out of UTF-8 mode. Unlike a dot, it always matches any line-ending characters. The feature is provided in Perl in order to match individual bytes in UTF-8 mode. Because it breaks up UTF-8 characters into individual bytes, what remains in the string may be a malformed UTF-8 string. For this reason, the \C escape sequence is best avoided.

PCRE does not allow \C to appear in lookbehind assertions (described below), because in UTF-8 mode this would make it impossible to calculate the length of the lookbehind.

SQUARE BRACKETS AND CHARACTER CLASSES

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. In UTF-8 mode, the character may occupy more than one byte. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lower case vowel, while [^aeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not. A class that starts with a circumflex

is not an assertion: it still consumes a character from the subject string, and therefore it fails if the current pointer is at the end of the string.

In UTF-8 mode, characters with values greater than 255 can be included in a class as a literal string of bytes, or by using the x escaping mechanism.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches "A" as well as "a", and a caseless [^aeiou] does not match "A", whereas a caseful version would. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.

Characters that might indicate line breaks are never treated in any special way when matching character classes, whatever line-ending sequence is in use, and whatever setting of the PCRE_DOTALL and PCRE_MULTILINE options is used. A class such as [^a] always matches one of these characters.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so [W-\]46] is interpreted as a class containing a range followed by two other characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example [\000-\037]. In UTF-8 mode, ranges can include characters whose values are greater than 255, for

example [$x{100}-x{2ff}$].

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W-c] is equivalent to [][\\^_`wxyzabc], matched caselessly, and in non-UTF-8 mode, if character tables for a French locale are in use, [\xc8-\xcb] matches accented E characters in both cases. In UTF-8 mode, PCRE supports the concept of case for characters with values greater than 128 only when it is compiled with Unicode property support.

The character types \d, \D, \p, \P, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [^\W_] matches any letter or digit, but not underscore.

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening square bracket (only when it can be interpreted as introducing a POSIX class name - see the next section), and the terminating closing square bracket. However, escaping other non-alphanumeric characters does no harm.

POSIX CHARACTER CLASSES

Perl supports the POSIX notation for character classes. This uses names enclosed by [: and :] within the enclosing square brackets. PCRE also supports this notation. For example,

[01[:alpha:]%]

matches "0", "1", any alphabetic character, or "%". The supported class names are

alnum	letters and digits
alpha	letters
ascii	character codes 0 - 127
blank	space or tab only
cntrl	control characters
digit	decimal digits (same as \d)
graph	printing characters, excluding space

lower	lower case letters
print	printing characters, including space
punct	printing characters, excluding letters and digits
space	white space (not quite the same as $\s)$
upper	upper case letters
word	"word" characters (same as \w)
xdigit	hexadecimal digits

The "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to \s, which does not include VT (for Perl compatibility).

The name "word" is a Perl extension, and "blank" is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a ^ character after the colon. For example,

[12[:^digit:]]

matches "1", "2", or any non-digit. PCRE (and Perl) also recognize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

In UTF-8 mode, characters with values greater than 128 do not match any of the POSIX character classes.

VERTICAL BAR

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

INTERNAL OPTION SETTING

The settings of the PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, and PCRE_EXTENDED options can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are

- i for PCRE_CASELESS
- m for PCRE_MULTILINE
- s for PCRE_DOTALL
- x for PCRE_EXTENDED

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im-sx), which sets PCRE_CASELESS and PCRE_MULTILINE while unsetting PCRE_DOTALL and PCRE_EXTENDED, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options (and it will therefore show up in data extracted by the **pcre_fullinfo()** function).

An option change within a subpattern (see below for a description of subpatterns) affects only that part of the current pattern that follows it, so

(a(?i)b)c

matches abc and aBc and no other strings (assuming PCRE_CASELESS is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

(a(?i)b|c)

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

The PCRE-specific options PCRE_DUPNAMES, PCRE_UNGREEDY, and PCRE_EXTRA can be changed in the same way as the Perl-compatible options

by using the characters J, U and X respectively.

SUBPATTERNS

Subpatterns are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

```
cat(aract|erpillar|)
```

matches one of the words "cat", "cataract", or "caterpillar". Without the parentheses, it would match "cataract", "erpillar" or an empty string.

2. It sets up the subpattern as a capturing subpattern. This means that, when the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the *ovector* argument of **pcre_exec()**. Opening parentheses are counted from left to right (starting from 1) to obtain numbers for the capturing subpatterns.

For example, if the string "the red king" is matched against the pattern

```
the ((red|white) (king|queen))
```

the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

```
the ((?:red|white) (king|queen))
```

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535.

As a convenient shorthand, if any option settings are required at the start of a

non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

(?i:saturday|sunday)
(?:(?i)saturday|sunday)

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

DUPLICATE SUBPATTERN NUMBERS

Perl 5.10 introduced a feature whereby each alternative in a subpattern uses the same numbers for its capturing parentheses. Such a subpattern starts with (?| and is itself a non-capturing subpattern. For example, consider this pattern:

(?|(Sat)ur|(Sun))day

Because the two alternatives are inside a (?| group, both sets of capturing parentheses are numbered one. Thus, when the pattern matches, you can look at captured substring number one, whichever alternative matched. This construct is useful when you want to capture part, but not all, of one of a number of alternatives. Inside a (?| group, parentheses are numbered as usual, but the number is reset at the start of each branch. The numbers of any capturing buffers that follow the subpattern start after the highest number used in any branch. The following example is taken from the Perl documentation. The numbers underneath show in which buffer the captured content will be stored.

# be	efore	branch-reset													after					
/ (a)	(?	х	(У)	z		(р	(q)	r)		(t)	u	(v))	(Ζ)	/x
# 1				2					2	3			2		3		4			

A backreference or a recursive call to a numbered subpattern always refers to the first one in the pattern with the given number.

An alternative approach to using this "branch reset" feature is to use duplicate named subpatterns, as described in the next section.

NAMED SUBPATTERNS

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE supports the naming of subpatterns. This feature was not added to Perl until release 5.10. Python had the feature earlier, and PCRE introduced it at release 4.0, using the Python syntax. PCRE now supports both the Perl and the Python syntax.

In PCRE, a subpattern can be named in one of three ways: (?<name>...) or (?'name'...) as in Perl, or (?P<name>...) as in Python. References to capturing parentheses from other parts of the pattern, such as <u>backreferences</u>, <u>recursion</u>, and <u>conditions</u>, can be made by name as well as by number.

Names consist of up to 32 alphanumeric characters and underscores. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present. The PCRE API provides function calls for extracting the name-to-number translation table from a compiled pattern. There is also a convenience function for extracting a captured substring by name.

By default, a name must be unique within a pattern, but it is possible to relax this constraint by setting the PCRE_DUPNAMES option at compile time. This can be useful for patterns where only one instance of the named parentheses can match. Suppose you want to match the name of a weekday, either as a 3-letter abbreviation or as the full name, and in both cases you want to extract the abbreviation. This pattern (ignoring the line breaks) does the job:

```
(?<DN>Mon|Fri|Sun)(?:day)?|
(?<DN>Tue)(?:sday)?|
(?<DN>Wed)(?:nesday)?|
(?<DN>Thu)(?:rsday)?|
(?<DN>Sat)(?:urday)?
```

There are five capturing substrings, but only one is ever set after a match. (An alternative way of solving this problem is to use a "branch reset" subpattern, as described in the previous section.)

The convenience function for extracting the data by name returns the substring for the first (and in this example, the only) subpattern of that name that matched. This saves searching to find which numbered subpattern it was. If you make a reference to a non-unique named subpattern from elsewhere in the pattern, the one that corresponds to the lowest number is used. For further details of the interfaces for handling named subpatterns, see the **<u>pcreapi</u>** documentation.

REPETITION

Repetition is specified by quantifiers, which can follow any of the following items:

```
a literal data character
the dot metacharacter
the \C escape sequence
the \X escape sequence (in UTF-8 mode with Unicode properties)
the \R escape sequence
an escape such as \d that matches a single character
a character class
a back reference (see next section)
a parenthesized subpattern (unless it is an assertion)
```

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

z{2,4}

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

[aeiou]{3,}

matches at least 3 successive vowels, but may match many more, while

\d{8}

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {,6} is not a quantifier, but a literal string of four characters.

In UTF-8 mode, quantifiers apply to UTF-8 characters rather than to individual bytes. Thus, for example, $x{100}{2}$ matches two UTF-8 characters, each of which is represented by a two-byte sequence. Similarly, when Unicode property

support is available, $X{3}$ matches three Unicode extended sequences, each of which may be several bytes long (and they may be of different lengths).

The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience, the three most common quantifiers have single-character abbreviations:

```
* is equivalent to {0,}
+ is equivalent to {1,}
? is equivalent to {0,1}
```

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

(a?)*

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */ and within the comment, individual * and / characters may appear. An attempt to match C comments by applying the pattern

/*.**/

to the string

/* first comment */ not comment /* second comment */

fails, because it matches the entire string owing to the greediness of the .* item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

/*.*?*/

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

\d??\d

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option is set (an option that is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or .{0,} and the PCRE_DOTALL option (equivalent to Perl's /s) is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally treats such a pattern as though it were preceded by \A.

In cases where it is known that the subject string contains no newlines, it is worth setting PCRE_DOTALL in order to obtain this optimization, or alternatively using ^ to indicate anchoring explicitly.

However, there is one situation where the optimization cannot be used. When .* is inside capturing parentheses that are the subject of a backreference elsewhere in the pattern, a match at the start may fail where a later one succeeds. Consider, for example:

(.*)abc\1

If the subject is "xyz123abc123" the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

When a capturing subpattern is repeated, the value captured is the substring that

matched the final iteration. For example, after

```
(tweedle[dume]{3}\s^*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

/(a|(b))+/

matches "aba" the value of the second captured substring is "b".

ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS

With both maximizing ("greedy") and minimizing ("ungreedy" or "lazy") repetition, failure of what follows normally causes the repeated item to be reevaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the subject line

123456bar

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and so on, before ultimately failing. "Atomic grouping" (a term taken from Jeffrey Friedl's book) provides the means for specifying that once a subpattern has matched, it is not to be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher gives up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with (?> as in this example:

(?>\d+)foo

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal. An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match in order to make the rest of the pattern match, (?>\d+) can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional + character following a quantifier. Using this notation, the previous example can be rewritten as

\d++foo

Possessive quantifiers are always greedy; the setting of the PCRE_UNGREEDY option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning of a possessive quantifier and the equivalent atomic group, though there may be a performance difference; possessive quantifiers should be slightly faster.

The possessive quantifier syntax is an extension to the Perl 5.8 syntax. Jeffrey Friedl originated the idea (and the name) in the first edition of his book. Mike McCloskey liked it, so implemented it when he built Sun's Java package, and PCRE copied it from there. It ultimately found its way into Perl at release 5.10.

PCRE has an optimization that automatically "possessifies" certain simple pattern constructs. For example, the sequence A+B is treated as A++B because there is no point in backtracking into a sequence of A's when B must follow.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

(\D+|<\d+>)*[!?]

matches an unlimited number of substrings that either consist of non-digits, or

digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

it takes a long time before reporting failure. This is because the string can be divided between the internal \D+ repeat and the external * repeat in a large number of ways, and all have to be tried. (The example uses [!?] rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

((?>\D+)|<\d+>)*[!?]

sequences of non-digits cannot be broken, and failure happens quickly.

BACK REFERENCES

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. A "forward back reference" of this type can make sense when a repetition is involved and the subpattern to the right has participated in an earlier iteration.

It is not possible to have a numerical "forward back reference" to a subpattern whose number is 10 or more using this syntax because a sequence such as \50 is interpreted as a character defined in octal. See the subsection entitled "Non-printing characters" <u>above</u> for further details of the handling of digits following a backslash. There is no such problem when named parentheses are used. A back reference to any subpattern is possible using named parentheses (see below).

Another way of avoiding the ambiguity inherent in the use of digits following a

backslash is to use the \g escape sequence, which is a feature introduced in Perl 5.10. This escape must be followed by a positive or a negative number, optionally enclosed in braces. These examples are all identical:

```
(ring), \1
(ring), \g1
(ring), \g{1}
```

A positive number specifies an absolute reference without the ambiguity that is present in the older syntax. It is also useful when literal digits follow the reference. A negative number is a relative reference. Consider this example:

```
(abc(def)ghi)\g{-1}
```

The sequence $g{-1}$ is a reference to the most recently started capturing subpattern before g, that is, is it equivalent to 2. Similarly, $g{-2}$ would be equivalent to 1. The use of relative references can be helpful in long patterns, and also in patterns that are created by joining together fragments that contain references within themselves.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see <u>"Subpatterns as subroutines"</u> below for a way of doing that). So the pattern

(sens|respons)e and \1ibility

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

((?i)rah)\s+\1

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

There are several different ways of writing back references to named subpatterns. The .NET syntax \k{name} and the Perl syntax \k<name> or \k'name' are supported, as is the Python syntax (?P=name). Perl 5.10's unified back reference syntax, in which \g can be used for both numeric and named references, is also supported. We could rewrite the above example in any of the following ways:

(?<p1>(?i)rah)\s+\k<p1>

(?'p1'(?i)rah)\s+\k{p1}
(?P<p1>(?i)rah)\s+(?P=p1)
(?<p1>(?i)rah)\s+\g{p1}

A subpattern that is referenced by name may appear in the pattern before or after the reference.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

(a|(bc))\2

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the PCRE_EXTENDED option is set, this can be whitespace. Otherwise an empty comment (see "Comments" below) can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

(a|b\1)+

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

ASSERTIONS

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as b, B, A, G, Z, z, A and are described <u>above</u>.

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it. An assertion subpattern is matched in the normal way, except that

it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Lookahead assertions

Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

\w+(?=;)

matches a word followed by a semicolon, but does not include the semicolon in the match, and

foo(?!bar)

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

(?!foo)bar

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (?!) because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

Lookbehind assertions

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

(?<!foo)bar

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several top-level alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (at least for 5.8), which requires all branches to match the same length of string. An assertion such as

(?<=ab(c|de))

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

(?<=abc|abde)

In some cases, the Perl 5.10 escape sequence \K <u>(see above)</u> can be used instead of a lookbehind assertion; this is not restricted to a fixed-length.

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed length and then try to match. If there are insufficient characters before the current position, the assertion fails.

PCRE does not allow the C escape (which matches a single byte in UTF-8 mode) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind. The X and R escapes, which can match different numbers of bytes, are also not permitted.

Possessive quantifiers can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

abcd\$

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

^.*abcd\$

the initial .* matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

^.*+(?<=abcd)

there can be no backtracking for the .*+ item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using multiple assertions

Several assertions (of any sort) may occur in succession. For example,

(?<=\d{3})(?<!999)foo

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does *not* match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is

(?<=\d{3}...)(?<!999)foo

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

(?<=(?<!foo)bar)baz

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

(?<=\d{3}(?!999)...)foo

is another pattern that matches "foo" preceded by three digits and any three characters that are not "999".

CONDITIONAL SUBPATTERNS

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are four kinds of condition: references to subpatterns, references to recursion, a pseudo-condition called DEFINE, and assertions.

Checking for a used subpattern by number

If the text between the parentheses consists of a sequence of digits, the condition is true if the capturing subpattern of that number has previously matched. An alternative notation is to precede the digits with a plus or minus sign. In this case, the subpattern number is relative rather than absolute. The most recently opened parentheses can be referenced by (?(-1), the next most recent by (?(-2), and so on. In looping constructs it can also make sense to refer to subsequent groups with constructs such as (?(+2).

Consider the following pattern, which contains non-significant white space to make it more readable (assume the PCRE_EXTENDED option) and to divide it into three parts for ease of discussion:

 $(\ (\)? [^{()}]+ (?(1) \))$

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If you were embedding this pattern in a larger one, you could use a relative reference:

...other stuff... (\()? $[^{()}]+ (?(-1) \))$...

This makes the fragment independent of the parentheses in the larger pattern.

Checking for a used subpattern by name

Perl uses the syntax (?(<name>)...) or (?('name')...) to test for a used subpattern by name. For compatibility with earlier versions of PCRE, which had this facility before Perl, the syntax (?(name)...) is also recognized. However, there is a possible ambiguity with this syntax, because subpattern names may consist entirely of digits. PCRE looks first for a named subpattern; if it cannot find one and the name consists entirely of digits, PCRE looks for a subpattern of that number, which must be greater than zero. Using subpattern names that consist entirely of digits is not recommended.

Rewriting the above example to use a named subpattern gives this:

(?<OPEN> \()? [^()]+ (?(<OPEN>) \))

Checking for pattern recursion

If the condition is the string (R), and there is no subpattern with the name R, the condition is true if a recursive call to the whole pattern or any subpattern has

been made. If digits or a name preceded by ampersand follow the letter R, for example:

(?(R3)...) or (?(R&name;)...)

the condition is true if the most recent recursion is into the subpattern whose number or name is given. This condition does not check the entire recursion stack.

At "top level", all these recursion test conditions are false. Recursive patterns are described below.

Defining subpatterns for use by reference only

If the condition is the string (DEFINE), and there is no subpattern with the name DEFINE, the condition is always false. In this case, there may be only one alternative in the subpattern. It is always skipped if control reaches this point in the pattern; the idea of DEFINE is that it can be used to define "subroutines" that can be referenced from elsewhere. (The use of "subroutines" is described below.) For example, a pattern to match an IPv4 address could be written like this (ignore whitespace and line breaks):

```
(?(DEFINE) (?<byte> 2[0-4]\d | 25[0-5] | 1\d\d | [1-9]?\d) )
\b (?&byte;) (\.(?&byte;)){3} \b
```

The first part of the pattern is a DEFINE group inside which a another group named "byte" is defined. This matches an individual component of an IPv4 address (a number less than 256). When matching takes place, this part of the pattern is skipped because DEFINE acts like a false condition.

The rest of the pattern uses references to the named group to match the four dotseparated components of an IPv4 address, insisting on a word boundary at each end.

Assertion conditions

If the condition is not in any of the above formats, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two

alternatives on the second line:

(?(?=[^a-z]*[a-z]) \d{2}-[a-z]{3}-\d{2} | \d{2}-\d{2}-\d{2})

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd, where aaa are letters and dd are digits.

COMMENTS

The sequence (?# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the PCRE_EXTENDED option is set, an unescaped # character outside a character class introduces a comment that continues to immediately after the next newline in the pattern.

RECURSIVE PATTERNS

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth.

For some time, Perl has provided a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern using code interpolation to solve the parentheses problem can be created like this:

```
re = qr{( (?: (?>[^()]+) | (?p{sre}))^* )}x;
```

The (?p{...}) item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears.

Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports special syntax for recursion of the entire pattern, and also for individual subpattern recursion. After its introduction in PCRE and Python, this kind of recursion was introduced into Perl at release 5.10.

A special item that consists of (? followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a "subroutine" call, which is described in the next section.) The special item (?R) or (?0) is a recursive call of the entire regular expression.

In PCRE (like Python, but unlike Perl), a recursive subpattern call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

This PCRE pattern solves the nested parentheses problem (assume the PCRE_EXTENDED option is set so that white space is ignored):

 $(((?>[^{()}]+) | (?R))*)$

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (that is, a correctly parenthesized substring). Finally there is a closing parenthesis.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

(\(((?>[^()]+) | (?1))* \))

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern.

In a larger pattern, keeping track of parenthesis numbers can be tricky. This is made easier by the use of relative references. (A Perl 5.10 feature.) Instead of (? 1) in the pattern above you can write (?-2) to refer to the second most recently opened parentheses preceding the recursion. In other words, a negative number counts capturing parentheses leftwards from the point at which it is encountered.

It is also possible to refer to subsequently opened parentheses, by writing

references such as (?+2). However, these cannot be recursive because the reference is not inside the parentheses that are referenced. They are always "subroutine" calls, as described in the next section.

An alternative approach is to use named parentheses instead. The Perl syntax for this is (?&name;); PCRE's earlier syntax (?P>name) is also supported. We could rewrite the above example as follows:

(?<pn> \(((?>[^()]+) | (?&pn;))* \))

If there is more than one subpattern with the same name, the earliest one is used.

This particular example pattern that we have been looking at contains nested unlimited repeats, and so the use of atomic grouping for matching strings of nonparentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

it yields "no match" quickly. However, if atomic grouping is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

At the end of a match, the values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If you want to obtain intermediate values, a callout function can be used (see below and the **pcrecallout** documentation). If the pattern above is matched against

(ab(cd)ef)

the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving

\((((?>[^()]+)	(?R))*)	\)
	۸					۸	
	۸					۸	

the string they capture is "ab(cd)ef", the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using **pcre_malloc**, freeing it via **pcre_free** afterwards. If no memory can be obtained, the match fails with the PCRE_ERROR_NOMEMORY error.

Do not confuse the (?R) item with the condition (R), which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

< (?: (?(R) \d++ | [^<>]*+) | (?R)) * >

In this pattern, (?(R) is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. The (?R) item is the actual recursive call.

SUBPATTERNS AS SUBROUTINES

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. The "called" subpattern may be defined before or after the reference. A numbered reference can be absolute or relative, as in these examples:

(...(absolute)...)...(?2)... (...(relative)...)...(?-1)... (...(?+1)...(relative)...

An earlier example pointed out that the pattern

(sens|respons)e and \1ibility

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

(sens|respons)e and (?1)ibility

is used, it does match "sense and responsibility" as well as the other two strings. Another example is given in the discussion of DEFINE above.

Like recursive subpatterns, a "subroutine" call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never reentered, even if it contains untried alternatives and there is a subsequent matching failure. When a subpattern is used as a subroutine, processing options such as caseindependence are fixed when the subpattern is defined. They cannot be changed for different calls. For example, consider this pattern:

(abc)(?i:(?-1))

It matches "abcabc". It does not match "abcABC" because the change of processing option does not affect the called subpattern.

CALLOUTS

Perl has a feature whereby using the sequence (?{...}) causes arbitrary Perl code to be obeyed in the middle of matching a regular expression. This makes it possible, amongst other things, to extract different substrings that match the same pair of parentheses when there is a repetition.

PCRE provides a similar feature, but of course it cannot obey arbitrary Perl code. The feature is called "callout". The caller of PCRE provides an external function by putting its entry point in the global variable *pcre_callout*. By default, this variable contains NULL, which disables all calling out.

Within a regular expression, (?C) indicates the points at which the external function is to be called. If you want to identify different callout points, you can put a number less than 256 after the letter C. The default value is zero. For example, this pattern has two callout points:

(?C1)abc(?C2)def

If the PCRE_AUTO_CALLOUT flag is passed to **pcre_compile()**, callouts are automatically installed before each item in the pattern. They are all numbered 255.

During matching, when PCRE reaches a callout point (and *pcre_callout* is set), the external function is called. It is provided with the number of the callout, the position in the pattern, and, optionally, one item of data originally supplied by the caller of **pcre_exec()**. The callout function may cause matching to proceed, to backtrack, or to fail altogether. A complete description of the interface to the callout function is given in the **pcrecallout** documentation.

SEE ALSO

pcreapi(3), pcrecallout(3), pcrematching(3), pcre(3).

AUTHOR

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 19 June 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcreperform man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

PCRE PERFORMANCE

Two aspects of performance are discussed below: memory usage and processing time. The way you express your pattern as a regular expression can affect both of them.

MEMORY USAGE

Patterns are compiled by PCRE into a reasonably efficient byte code, so that most simple patterns do not use much memory. However, there is one case where memory usage can be unexpectedly large. When a parenthesized subpattern has a quantifier with a minimum greater than 1 and/or a limited maximum, the whole subpattern is repeated in the compiled code. For example, the pattern

(abc|def){2,4}

is compiled as if it were

(abc|def)(abc|def)((abc|def)(abc|def)?)?

(Technical aside: It is done this way so that backtrack points within each of the repetitions can be independently maintained.)

For regular expressions whose quantifiers use only small numbers, this is not usually a problem. However, if the numbers are large, and particularly if such repetitions are nested, the memory usage can become an embarrassment. For example, the very simple pattern

((ab){1,1000}c){1,3}

uses 51K bytes when compiled. When PCRE is compiled with its default internal pointer size of two bytes, the size limit on a compiled pattern is 64K, and this is reached with the above pattern if the outer repetition is increased from 3 to 4. PCRE can be compiled to use larger internal pointers and thus handle larger compiled patterns, but it is better to try to rewrite your pattern to use less memory if you can.

One way of reducing the memory usage for such patterns is to make use of PCRE's <u>"subroutine"</u> facility. Re-writing the above pattern as

((ab)(?2){0,999}c)(?1){0,2}

reduces the memory requirements to 18K, and indeed it remains under 20K even with the outer repetition increased to 100. However, this pattern is not exactly equivalent, because the "subroutine" calls are treated as <u>atomic groups</u> into which there can be no backtracking if there is a subsequent matching failure. Therefore, PCRE cannot do this kind of rewriting automatically. Furthermore, there is a noticeable loss of speed when executing the modified pattern. Nevertheless, if the atomic grouping is not a problem and the loss of speed is acceptable, this kind of rewriting will allow you to process patterns that PCRE cannot otherwise handle.

PROCESSING TIME

Certain items in regular expression patterns are processed more efficiently than others. It is more efficient to use a character class like [aeiou] than a set of single-character alternatives such as (a|e|i|o|u). In general, the simplest construction that provides the required behaviour is usually the most efficient. Jeffrey Friedl's book contains a lot of useful general discussion about optimizing regular expressions for efficient performance. This document contains a few observations about PCRE.

Using Unicode character properties (the \p, \P, and \X escapes) is slow, because PCRE has to scan a structure that contains data for over fifteen thousand characters whenever it needs a character's property. If you can find an alternative pattern that does not use character properties, it will probably be faster.

When a pattern begins with .* not in parentheses, or in parentheses that are not the subject of a backreference, and the PCRE_DOTALL option is set, the pattern

is implicitly anchored by PCRE, since it can match only at the start of a subject string. However, if PCRE_DOTALL is not set, PCRE cannot make this optimization, because the . metacharacter does not then match a newline, and if the subject string contains newlines, the pattern may match from the character immediately following one of them instead of from the very start. For example, the pattern

.*second

matches the subject "first\nand second" (where \n stands for a newline character), with the match starting at the seventh character. In order to do this, PCRE has to retry the match starting after every newline in the subject.

If you are using such a pattern with subject strings that do not contain newlines, the best performance is obtained by setting PCRE_DOTALL, or starting the pattern with \land .* or \land .*? to indicate explicit anchoring. That saves PCRE from having to scan along the subject looking for a newline to restart at.

Beware of patterns that contain nested indefinite repeats. These can take a long time to run when applied to a string that does not match. Consider the pattern fragment

^(a+)*

This can match "aaaa" in 16 different ways, and this number increases very rapidly as the string gets longer. (The * repeat can match 0, 1, 2, 3, or 4 times, and for each of those cases other than 0 or 4, the + repeats can match different numbers of times.) When the remainder of the pattern is such that the entire match is going to fail, PCRE has in principle to try every possible variation, and this can take an extremely long time, even for relatively short strings.

An optimization catches some of the more simple cases such as

(a+)*b

where a literal character follows. Before embarking on the standard matching procedure, PCRE checks that there is a "b" later in the subject string, and if there is not, it fails the match immediately. However, when there is no following literal this optimization cannot be used. You can see the difference by comparing the behaviour of

(a+)*\d

with the pattern above. The former gives a failure almost instantly when applied to a whole line of "a" characters, whereas the latter takes an appreciable time with strings longer than about 20 characters.

In many cases, the solution to this kind of performance issue is to use an atomic group or a possessive quantifier.

AUTHOR

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 06 March 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcreposix man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- SYNOPSIS OF POSIX API
- **DESCRIPTION**
- <u>COMPILING A PATTERN</u>
- MATCHING NEWLINE CHARACTERS
- MATCHING A PATTERN
- ERROR MESSAGES
- <u>MEMORY USAGE</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

SYNOPSIS OF POSIX API

#include <pcreposix.h>

int regcomp(regex_t *preg, const char *pattern, int cflags);

int regexec(regex_t *preg, const char *string, size_t nmatch, regmatch_t
pmatch[], int eflags);

size_t regerror(int errcode, const regex_t *preg, char *errbuf, size_t
errbuf_size);

void regfree(regex_t *preg);

DESCRIPTION

This set of functions provides a POSIX-style API to the PCRE regular expression package. See the **pcreapi** documentation for a description of PCRE's

native API, which contains much additional functionality.

The functions described here are just wrapper functions that ultimately call the PCRE native API. Their prototypes are defined in the **pcreposix.h** header file, and on Unix systems the library itself is called **pcreposix.a**, so can be accessed by adding **-lpcreposix** to the command for linking an application that uses them. Because the POSIX functions call the native ones, it is also necessary to add **-lpcre**.

I have implemented only those option bits that can be reasonably mapped to PCRE native options. In addition, the option REG_EXTENDED is defined with the value zero. This has no effect, but since programs that are written to the POSIX interface often use it, this makes it easier to slot in PCRE as a replacement library. Other POSIX options are not even defined.

When PCRE is called via these functions, it is only the API that is POSIX-like in style. The syntax and semantics of the regular expressions themselves are still those of Perl, subject to the setting of various PCRE options, as described below. "POSIX-like in style" means that the API approximates to the POSIX definition; it is not fully POSIX-compatible, and in multi-byte encoding domains it is probably even less compatible.

The header for these functions is supplied as **pcreposix.h** to avoid any potential clash with other POSIX libraries. It can, of course, be renamed or aliased as **regex.h**, which is the "correct" name. It provides two structure types, *regex_t* for compiled internal forms, and *regmatch_t* for returning captured substrings. It also defines some constants whose names start with "REG_"; these are used for setting options and identifying error codes.

COMPILING A PATTERN

The function **regcomp()** is called to compile a pattern into an internal form. The pattern is a C string terminated by a binary zero, and is passed in the argument *pattern*. The *preg* argument is a pointer to a **regex_t** structure that is used as a base for storing information about the compiled regular expression.

The argument *cflags* is either zero, or contains one or more of the bits defined by the following macros:

REG_DOTALL

The PCRE_DOTALL option is set when the regular expression is passed for compilation to the native function. Note that REG_DOTALL is not part of the POSIX standard.

REG_ICASE

The PCRE_CASELESS option is set when the regular expression is passed for compilation to the native function.

REG_NEWLINE

The PCRE_MULTILINE option is set when the regular expression is passed for compilation to the native function. Note that this does *not* mimic the defined POSIX behaviour for REG_NEWLINE (see the following section).

REG_NOSUB

The PCRE_NO_AUTO_CAPTURE option is set when the regular expression is passed for compilation to the native function. In addition, when a pattern that is compiled with this flag is passed to **regexec()** for matching, the *nmatch* and *pmatch* arguments are ignored, and no captured strings are returned.

REG_UTF8

The PCRE_UTF8 option is set when the regular expression is passed for compilation to the native function. This causes the pattern itself and all data strings used for matching it to be treated as UTF-8 strings. Note that REG_UTF8 is not part of the POSIX standard.

In the absence of these flags, no options are passed to the native function. This means the the regex is compiled with PCRE default semantics. In particular, the way it handles newline characters in the subject string is the Perl way, not the POSIX way. Note that setting PCRE_MULTILINE has only *some* of the effects specified for REG_NEWLINE. It does not affect the way newlines are matched by . (they aren't) or by a negative class such as [^a] (they are).

The yield of **regcomp()** is zero on success, and non-zero otherwise. The *preg* structure is filled in on success, and one member of the structure is public: *re_nsub* contains the number of capturing subpatterns in the regular expression. Various error codes are defined in the header file.

MATCHING NEWLINE CHARACTERS

This area is not simple, because POSIX and Perl take different views of things. It is not possible to get PCRE to obey POSIX semantics, but then PCRE was never intended to be a POSIX engine. The following table lists the different possibilities for matching newline characters in PCRE:

	Default	Change with
<pre>. matches newline</pre>	no	PCRE_DOTALL
newline matches [^a]	yes	not changeable
\$ matches \n at end	yes	PCRE_DOLLARENDONLY
\$ matches \n in middle	no	PCRE_MULTILINE
^ matches \n in middle	no	PCRE_MULTILINE

This is the equivalent table for POSIX:

	Default	Change with
<pre>. matches newline</pre>	yes	REG_NEWLINE
newline matches [^a]	yes	REG_NEWLINE
\$ matches \n at end	no	REG_NEWLINE
\$ matches \n in middle	no	REG_NEWLINE
^ matches \n in middle	no	REG_NEWLINE

PCRE's behaviour is the same as Perl's, except that there is no equivalent for PCRE_DOLLAR_ENDONLY in Perl. In both PCRE and Perl, there is no way to stop newline from matching [^a].

The default POSIX newline handling can be obtained by setting PCRE_DOTALL and PCRE_DOLLAR_ENDONLY, but there is no way to make PCRE behave exactly as for the REG_NEWLINE action.

MATCHING A PATTERN

The function **regexec()** is called to match a compiled pattern *preg* against a given *string*, which is terminated by a zero byte, subject to the options in *eflags*. These can be:

REG_NOTBOL

The PCRE_NOTBOL option is set when calling the underlying PCRE matching

function.

REG_NOTEOL

The PCRE_NOTEOL option is set when calling the underlying PCRE matching function.

If the pattern was compiled with the REG_NOSUB flag, no data about any matched strings is returned. The *nmatch* and *pmatch* arguments of **regexec()** are ignored.

Otherwise, the portion of the string that was matched, and also any captured substrings, are returned via the *pmatch* argument, which points to an array of *nmatch* structures of type *regmatch_t*, containing the members *rm_so* and *rm_eo*. These contain the offset to the first character of each substring and the offset to the first character after the end of each substring, respectively. The 0th element of the vector relates to the entire portion of *string* that was matched; subsequent elements relate to the capturing subpatterns of the regular expression. Unused entries in the array have both structure members set to -1.

A successful match yields a zero return; various error codes are defined in the header file, of which REG_NOMATCH is the "expected" failure code.

ERROR MESSAGES

The **regerror()** function maps a non-zero errorcode from either **regcomp()** or **regexec()** to a printable message. If *preg* is not NULL, the error should have arisen from the use of that structure. A message terminated by a binary zero is placed in *errbuf*. The length of the message, including the zero, is limited to *errbuf_size*. The yield of the function is the size of buffer needed to hold the whole message.

MEMORY USAGE

Compiling a regular expression causes memory to be allocated and associated with the *preg* structure. The function **regfree()** frees all such memory, after which *preg* may no longer be used as a compiled expression.

AUTHOR

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 06 March 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcreprecompile man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- <u>SAVING AND RE-USING PRECOMPILED PCRE PATTERNS</u>
- <u>SAVING A COMPILED PATTERN</u>
- <u>RE-USING A PRECOMPILED PATTERN</u>
- <u>COMPATIBILITY WITH DIFFERENT PCRE RELEASES</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

SAVING AND RE-USING PRECOMPILED PCRE PATTERNS

If you are running an application that uses a large number of regular expression patterns, it may be useful to store them in a precompiled form instead of having to compile them every time the application is run. If you are not using any private character tables (see the <u>pcre_maketables()</u> documentation), this is relatively straightforward. If you are using private tables, it is a little bit more complicated.

If you save compiled patterns to a file, you can copy them to a different host and run them there. This works even if the new host has the opposite endianness to the one on which the patterns were compiled. There may be a small performance penalty, but it should be insignificant. However, compiling regular expressions with one version of PCRE for use with a different version is not guaranteed to work and may cause crashes.

SAVING A COMPILED PATTERN

The value returned by **pcre_compile()** points to a single block of memory that holds the compiled pattern and associated data. You can find the length of this

block in bytes by calling pcre_fullinfo() with an argument of

PCRE_INFO_SIZE. You can then save the data in any appropriate manner. Here is sample code that compiles a pattern and writes it to a file. It assumes that the variable *fd* refers to a file that is open for output:

```
int erroroffset, rc, size;
char *error;
pcre *re;
re = pcre_compile("my pattern", 0, &error;, &erroroffset;, NULL);
if (re == NULL) { ... handle errors ... }
rc = pcre_fullinfo(re, NULL, PCRE_INF0_SIZE, &size;);
if (rc < 0) { ... handle errors ... }
rc = fwrite(re, 1, size, fd);
if (rc != size) { ... handle errors ... }
```

In this example, the bytes that comprise the compiled pattern are copied exactly. Note that this is binary data that may contain any of the 256 possible byte values. On systems that make a distinction between binary and non-binary data, be sure that the file is opened for binary output.

If you want to write more than one pattern to a file, you will have to devise a way of separating them. For binary data, preceding each pattern with its length is probably the most straightforward approach. Another possibility is to write out the data in hexadecimal instead of binary, one pattern to a line.

Saving compiled patterns in a file is only one possible way of storing them for later use. They could equally well be saved in a database, or in the memory of some daemon process that passes them via sockets to the processes that want them.

If the pattern has been studied, it is also possible to save the study data in a similar way to the compiled pattern itself. When studying generates additional information, **pcre_study()** returns a pointer to a **pcre_extra** data block. Its format is defined in the <u>section on matching a pattern</u> in the <u>pcreapi</u> documentation. The *study_data* field points to the binary study data, and this is what you must save (not the **pcre_extra** block itself). The length of the study data can be obtained by calling **pcre_fullinfo()** with an argument of PCRE_INFO_STUDYSIZE. Remember to check that **pcre_study()** did return a non-NULL value before trying to save the study data.

RE-USING A PRECOMPILED PATTERN

Re-using a precompiled pattern is straightforward. Having reloaded it into main memory, you pass its pointer to **pcre_exec()** or **pcre_dfa_exec()** in the usual way. This should work even on another host, and even if that host has the opposite endianness to the one where the pattern was compiled.

However, if you passed a pointer to custom character tables when the pattern was compiled (the *tableptr* argument of **pcre_compile()**), you must now pass a similar pointer to **pcre_exec()** or **pcre_dfa_exec()**, because the value saved with the compiled pattern will obviously be nonsense. A field in a **pcre_extra()** block is used to pass this data, as described in the <u>section on matching a pattern</u> in the **pcreapi** documentation.

If you did not provide custom character tables when the pattern was compiled, the pointer in the compiled pattern is NULL, which causes **pcre_exec()** to use PCRE's internal tables. Thus, you do not need to take any special action at run time in this case.

If you saved study data with the compiled pattern, you need to create your own **pcre_extra** data block and set the *study_data* field to point to the reloaded study data. You must also set the PCRE_EXTRA_STUDY_DATA bit in the *flags* field to indicate that study data is present. Then pass the **pcre_extra** block to **pcre_exec()** or **pcre_dfa_exec()** in the usual way.

COMPATIBILITY WITH DIFFERENT PCRE RELEASES

In general, it is safest to recompile all saved patterns when you update to a new PCRE release, though not all updates actually require this. Recompiling is definitely needed for release 7.2.

<u>AUTHOR</u>

Philip Hazel University Computing Service Cambridge CB2 3QH, England. **REVISION**

Last updated: 13 June 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcresample man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

PCRE SAMPLE PROGRAM

A simple, complete demonstration program, to get you started with using PCRE, is supplied in the file *pcredemo.c* in the PCRE distribution.

The program compiles the regular expression that is its first argument, and matches it against the subject string in its second argument. No PCRE options are set, and default character tables are used. If matching succeeds, the program outputs the portion of the subject that matched, together with the contents of any captured substrings.

If the -g option is given on the command line, the program then goes on to check for further matches of the same regular expression in the same subject string. The logic is a little bit tricky because of the possibility of matching an empty string. Comments in the code explain what is going on.

The demonstration program is automatically built if you use "./configure;make" to build PCRE. Otherwise, if PCRE is installed in the standard include and library directories for your system, you should be able to compile the demonstration program using this command:

```
gcc -o pcredemo pcredemo.c -lpcre
```

If PCRE is installed elsewhere, you may need to add additional options to the command line. For example, on a Unix-like system that has PCRE installed in */usr/local*, you can compile the demonstration program using a command like this:

gcc -o pcredemo -I/usr/local/include pcredemo.c -L/usr/local/lib -

Once you have compiled the demonstration program, you can run simple tests like this:

./pcredemo 'cat|dog' 'the cat sat on the mat'
./pcredemo -g 'cat|dog' 'the dog sat on the cat'

Note that there is a much more comprehensive test program, called **pcretest**, which supports many more facilities for testing regular expressions and the PCRE library. The **pcredemo** program is provided as a simple coding example.

On some operating systems (e.g. Solaris), when PCRE is not installed in the standard library directory, you may get an error like this when you try to run **pcredemo**:

```
ld.so.1: a.out: fatal: libpcre.so.0: open failed: No such file or
```

This is caused by the way shared library support works on those systems. You need to add

-R/usr/local/lib

(for example) to the compile command to get round this problem.

AUTHOR

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 13 June 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcrestack man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

PCRE DISCUSSION OF STACK USAGE

When you call **pcre_exec()**, it makes use of an internal function called **match()**. This calls itself recursively at branch points in the pattern, in order to remember the state of the match so that it can back up and try a different alternative if the first one fails. As matching proceeds deeper and deeper into the tree of possibilities, the recursion depth increases.

Not all calls of **match()** increase the recursion depth; for an item such as a* it may be called several times at the same level, after matching different numbers of a's. Furthermore, in a number of cases where the result of the recursive call would immediately be passed back as the result of the current call (a "tail recursion"), the function is just restarted instead.

The **pcre_dfa_exec()** function operates in an entirely different way, and hardly uses recursion at all. The limit on its complexity is the amount of workspace it is given. The comments that follow do NOT apply to **pcre_dfa_exec()**; they are relevant only for **pcre_exec()**.

You can set limits on the number of times that **match()** is called, both in total and recursively. If the limit is exceeded, an error occurs. For details, see the <u>section on extra data for pcre_exec()</u> in the <u>pcreapi</u> documentation.

Each time that **match()** is actually called recursively, it uses memory from the process stack. For certain kinds of pattern and data, very large amounts of stack may be needed, despite the recognition of "tail recursion". You can often reduce the amount of recursion, and therefore the amount of stack used, by modifying the pattern that is being matched. Consider, for example, this pattern:

([^<]|<(?!inet))+

It matches from wherever it starts until it encounters "<inet" or the end of the data, and is the kind of pattern that might be used when processing an XML file. Each iteration of the outer parentheses matches either one character that is not "<" or a "<" that is not followed by "inet". However, each time a parenthesis is processed, a recursion occurs, so this formulation uses a stack frame for each matched character. For a long string, a lot of stack is required. Consider now this rewritten pattern, which matches exactly the same strings:

([^<]++|<(?!inet))+

This uses very much less stack, because runs of characters that do not contain " <" are "swallowed" in one item inside the parentheses. Recursion happens only when a "<" character that is not followed by "inet" is encountered (and we assume this is relatively rare). A possessive quantifier is used to stop any backtracking into the runs of non-"<" characters, but that is not related to stack usage.

This example shows that one way of avoiding stack problems when matching long subject strings is to write repeated parenthesized subpatterns to match more than one character whenever possible.

In environments where stack memory is constrained, you might want to compile PCRE to use heap memory instead of stack for remembering back-up points. This makes it run a lot more slowly, however. Details of how to do this are given in the **pcrebuild** documentation. When built in this way, instead of using the stack, PCRE obtains and frees memory by calling the functions that are pointed to by the **pcre_stack_malloc** and **pcre_stack_free** variables. By default, these point to **malloc()** and **free()**, but you can replace the pointers to cause PCRE to use your own functions. Since the block sizes are always the same, and are always freed in reverse order, it may be possible to implement customized memory handlers that are more efficient than the standard functions.

In Unix-like environments, there is not often a problem with the stack unless very long strings are involved, though the default limit on stack size varies from system to system. Values from 8Mb to 64Mb are common. You can find your default limit by running the command:

ulimit -s

Unfortunately, the effect of running out of stack is often SIGSEGV, though sometimes a more explicit error message is given. You can normally increase the limit on stack size by code such as this:

struct rlimit rlim; getrlimit(RLIMIT_STACK, &rlim;); rlim.rlim_cur = 100*1024*1024; setrlimit(RLIMIT_STACK, &rlim;);

This reads the current limits (soft and hard) using **getrlimit()**, then attempts to increase the soft limit to 100Mb using **setrlimit()**. You must do this before calling **pcre_exec()**.

PCRE has an internal counter that can be used to limit the depth of recursion, and thus cause **pcre_exec()** to give an error code before it runs out of stack. By default, the limit is very large, and unlikely ever to operate. It can be changed when PCRE is built, and it can also be set when **pcre_exec()** is called. For details of these interfaces, see the **pcrebuild** and **pcreapi** documentation.

As a very rough rule of thumb, you should reckon on about 500 bytes per recursion. Thus, if you want to limit your stack usage to 8Mb, you should set the limit at 16000 recursions. A 64Mb stack, on the other hand, can support around 128000 recursions. The **pcretest** test program has a command line option (**-S**) that can be used to increase the size of its stack.

AUTHOR

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 05 June 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.

pcretest man page

Return to the <u>PCRE index page</u>.

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- <u>SYNOPSIS</u>
- <u>OPTIONS</u>
- <u>DESCRIPTION</u>
- PATTERN MODIFIERS
- DATA LINES
- THE ALTERNATIVE MATCHING FUNCTION
- DEFAULT OUTPUT FROM PCRETEST
- OUTPUT FROM THE ALTERNATIVE MATCHING FUNCTION
- **RESTARTING AFTER A PARTIAL MATCH**
- <u>CALLOUTS</u>
- <u>NON-PRINTING CHARACTERS</u>
- <u>SAVING AND RELOADING COMPILED PATTERNS</u>
- <u>SEE ALSO</u>
- <u>AUTHOR</u>
- <u>REVISION</u>

SYNOPSIS

pcretest [options] [source] [destination]

pcretest was written as a test program for the PCRE regular expression library itself, but it can also be used for experimenting with regular expressions. This document describes the features of the test program; for details of the regular expressions themselves, see the **pcrepattern** documentation. For details of the PCRE library function calls and their options, see the **pcreapi** documentation.

OPTIONS

-b Behave as if each regex has the **/B** (show bytecode) modifier; the internal form is output after compilation.

-C Output the version number of the PCRE library, and all available information about the optional features that are included, and then exit.

-**d** Behave as if each regex has the /**D** (debug) modifier; the internal form and information about the compiled pattern is output after compilation; -**d** is equivalent to -**b** -**i**.

-dfa Behave as if each data line contains the \D escape sequence; this causes the alternative matching function, **pcre_dfa_exec()**, to be used instead of the standard **pcre_exec()** function (more detail is given below).

-help Output a brief summary these options and then exit.

-i Behave as if each regex has the /**I** modifier; information about the compiled pattern is given after compilation.

-m Output the size of each compiled pattern after it has been compiled. This is equivalent to adding /**M** to each regular expression. For compatibility with earlier versions of pcretest, **-s** is a synonym for **-m**.

-o osize Set the number of elements in the output vector that is used when calling pcre_exec() or pcre_dfa_exec() to be osize. The default value is 45, which is enough for 14 capturing subexpressions for pcre_exec() or 22 different matches for pcre_dfa_exec(). The vector size can be changed for individual matching calls by including \O in the data line (see below).

-p Behave as if each regex has the **/P** modifier; the POSIX wrapper API is used to call PCRE. None of the other options has any effect when **-p** is set.

-q Do not output the version number of **pcretest** at the start of execution.

-S size On Unix-like systems, set the size of the runtime stack to size megabytes.

-t Run each compile, study, and match many times with a timer, and output resulting time per compile or match (in milliseconds). Do not set **-m** with **-t**, because you will then get the size output a zillion times, and the timing will be distorted. You can control the number of iterations that are used for timing by

following **-t** with a number (as a separate item on the command line). For example, "-t 1000" would iterate 1000 times. The default is to iterate 500000 times.

-tm This is like **-t** except that it times only the matching phase, not the compile or study phases.

DESCRIPTION

If **pcretest** is given two filename arguments, it reads from the first and writes to the second. If it is given only one filename argument, it reads from that file and writes to stdout. Otherwise, it reads from stdin and writes to stdout, and prompts for each line of input, using "re>" to prompt for regular expressions, and "data>" to prompt for data lines.

The program handles any number of sets of input on a single input file. Each set starts with a regular expression, and continues with any number of data lines to be matched against the pattern.

Each data line is matched separately and independently. If you want to do multiline matches, you have to use the \n escape sequence (or \r or \r\n, etc., depending on the newline setting) in a single line of input to encode the newline sequences. There is no limit on the length of data lines; the input buffer is automatically extended if it is too small.

An empty line signals the end of the data lines, at which point a new regular expression is read. The regular expressions are given enclosed in any non-alphanumeric delimiters other than backslash, for example:

/(a|bc)x+yz/

White space before the initial delimiter is ignored. A regular expression may be continued over several input lines, in which case the newline characters are included within it. It is possible to include the delimiter within the pattern by escaping it, for example

```
/abc\/def/
```

If you do so, the escape and the delimiter form part of the pattern, but since delimiters are always non-alphanumeric, this does not affect its interpretation. If

the terminating delimiter is immediately followed by a backslash, for example,

/abc/\

then a backslash is added to the end of the pattern. This is done to provide a way of testing the error condition that arises if a pattern finishes with a backslash, because

/abc\/

is interpreted as the first line of a pattern that starts with "abc/", causing pcretest to read the next line as a continuation of the regular expression.

PATTERN MODIFIERS

A pattern may be followed by any number of modifiers, which are mostly single characters. Following Perl usage, these are referred to below as, for example, "the /i modifier", even though the delimiter of the pattern need not always be a slash, and no slash is used when writing modifiers. Whitespace may appear between the final pattern delimiter and the first modifier, and between the modifiers themselves.

The /i, /m, /s, and /x modifiers set the PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, or PCRE_EXTENDED options, respectively, when **pcre_compile()** is called. These four modifier letters have the same effect as they do in Perl. For example:

/caseless/i

The following table shows additional modifiers for setting PCRE options that do not correspond to anything in Perl:

/A	PCRE_ANCHORED
/C	PCRE_AUT0_CALLOUT
/E	PCRE_DOLLAR_ENDONLY
/f	PCRE_FIRSTLINE
/J	PCRE_DUPNAMES
/N	PCRE_NO_AUTO_CAPTURE
/U	PCRE_UNGREEDY
/X	PCRE_EXTRA
/ <cr></cr>	PCRE_NEWLINE_CR
/ <lf></lf>	PCRE_NEWLINE_LF
/ <crlf></crlf>	PCRE_NEWLINE_CRLF

/<anycrlf> PCRE_NEWLINE_ANYCRLF
/<any> PCRE_NEWLINE_ANY

Those specifying line ending sequencess are literal strings as shown. This example sets multiline matching with CRLF as the line ending sequence:

/^abc/m<crlf>

Details of the meanings of these PCRE options are given in the **<u>pcreapi</u>** documentation.

Finding all matches in a string

Searching for all possible matches within each subject string can be requested by the /**g** or /**G** modifier. After finding a match, PCRE is called again to search the remainder of the subject string. The difference between /**g** and /**G** is that the former uses the *startoffset* argument to **pcre_exec()** to start searching at a new point within the entire string (which is in effect what Perl does), whereas the latter passes over a shortened substring. This makes a difference to the matching process if the pattern begins with a lookbehind assertion (including \b or \B).

If any call to **pcre_exec()** in a /**g** or /**G** sequence matches an empty string, the next call is done with the PCRE_NOTEMPTY and PCRE_ANCHORED flags set in order to search for another, non-empty, match at the same point. If this second match fails, the start offset is advanced by one, and the normal match is retried. This imitates the way Perl handles such cases when using the /**g** modifier or the **split()** function.

Other modifiers

There are yet more modifiers for controlling the way **pcretest** operates.

The /+ modifier requests that as well as outputting the substring that matched the entire pattern, pcretest should in addition output the remainder of the subject string. This is useful for tests where the subject contains multiple copies of the same substring.

The **/B** modifier is a debugging feature. It requests that **pcretest** output a representation of the compiled byte code after compilation. Normally this

information contains length and offset values; however, if $/\mathbb{Z}$ is also present, this data is replaced by spaces. This is a special feature for use in the automatic test scripts; it ensures that the same output is generated for different internal link sizes.

The /L modifier must be followed directly by the name of a locale, for example,

/pattern/Lfr_FR

For this reason, it must be the last modifier. The given locale is set, **pcre_maketables()** is called to build a set of character tables for the locale, and this is then passed to **pcre_compile()** when compiling the regular expression. Without an /L modifier, NULL is passed as the tables pointer; that is, /L applies only to the expression on which it appears.

The /I modifier requests that **pcretest** output information about the compiled pattern (whether it is anchored, has a fixed first character, and so on). It does this by calling **pcre_fullinfo()** after compiling a pattern. If the pattern is studied, the results of that are also output.

The /**D** modifier is a PCRE debugging feature, and is equivalent to /**BI**, that is, both the /**B** and the /**I** modifiers.

The /**F** modifier causes **pcretest** to flip the byte order of the fields in the compiled pattern that contain 2-byte and 4-byte numbers. This facility is for testing the feature in PCRE that allows it to execute patterns that were compiled on a host with a different endianness. This feature is not available when the POSIX interface to PCRE is being used, that is, when the /**P** pattern modifier is specified. See also the section about saving and reloading compiled patterns below.

The **/S** modifier causes **pcre_study()** to be called after the expression has been compiled, and the results used when the expression is matched.

The $/\mathbf{M}$ modifier causes the size of memory block used to hold the compiled pattern to be output.

The **/P** modifier causes **pcretest** to call PCRE via the POSIX wrapper API rather than its native API. When this is done, all other modifiers except **/i**, **/m**, and **/+** are ignored. REG_ICASE is set if **/i** is present, and REG_NEWLINE is set if **/m**

is present. The wrapper functions force PCRE_DOLLAR_ENDONLY always, and PCRE_DOTALL unless REG_NEWLINE is set.

The **/8** modifier causes **pcretest** to call PCRE with the PCRE_UTF8 option set. This turns on support for UTF-8 character handling in PCRE, provided that it was compiled with this support enabled. This modifier also causes any non-printing characters in output strings to be printed using the $x{hh...}$ notation if they are valid UTF-8 sequences.

If the **/?** modifier is used with **/8**, it causes **pcretest** to call **pcre_compile()** with the PCRE_NO_UTF8_CHECK option, to suppress the checking of the string for UTF-8 validity.

DATA LINES

Before each data line is passed to **pcre_exec()**, leading and trailing whitespace is removed, and it is then scanned for \ escapes. Some of these are pretty esoteric features, intended for checking out some of the more complicated features of PCRE. If you are just testing "ordinary" regular expressions, you probably don't need any of these. The following escapes are recognized:

\a	alarm (BEL, $\times 07$)
\b	backspace (\x08)
\e	escape (\x27)
\f	formfeed (\x0c)
\n	newline (\x0a)
\qdd	set the PCRE_MATCH_LIMIT limit to dd (any number of dig
١r	carriage return (\x0d)
\t	tab (\x09)
\v	vertical tab (\x0b)
\nnn	octal character (up to 3 octal digits)
\xhh	hexadecimal character (up to 2 hex digits)
$x{hh}$	hexadecimal character, any number of digits in UTF-8 mc
\A	pass the PCRE_ANCHORED option to pcre_exec() or pcre_df
∖В	pass the PCRE_NOTBOL option to pcre_exec() or pcre_dfa _
\Cdd	call pcre_copy_substring() for substring dd after a suc
\Cname	<pre>call pcre_copy_named_substring() for substring "name" a</pre>
	ated by next non alphanumeric character)
\C+	show the current captured substrings at callout time
\C-	do not supply a callout function
\C!n	return 1 instead of 0 when callout number n is reached
\C!n!m	return 1 instead of 0 when callout number n is reached
\C*n	pass the number n (may be negative) as callout data; th

\D \F \Gdd \Gname	<pre>use the pcre_dfa_exec() match function only shortest match for pcre_dfa_exec() call pcre_get_substring() for substring dd after a succ call pcre_get_named_substring() for substring "name" af</pre>
١L	call pcre_get_substringlist() after a successful match
λM	discover the minimum MATCH_LIMIT and MATCH_LIMIT_RECURS
\N	pass the PCRE_NOTEMPTY option to pcre_exec() or pcre_df
\0dd	set the size of the output vector passed to pcre_exec()
\Ρ	pass the PCRE_PARTIAL option to pcre_exec() or pcre_dfa
\Qdd	set the PCRE_MATCH_LIMIT_RECURSION limit to dd (any num
\R	pass the PCRE_DFA_RESTART option to pcre_dfa_exec()
\S	output details of memory get/free calls during matching
λZ	<pre>pass the PCRE_NOTEOL option to pcre_exec() or pcre_dfa_</pre>
\?	pass the PCRE_NO_UTF8_CHECK option to pcre_exec() or pc
∖>dd	start the match at offset dd (any number of digits);
	this sets the <i>startoffset</i> argument for pcre_exec() or
\ <cr></cr>	<pre>pass the PCRE_NEWLINE_CR option to pcre_exec() or pcre_</pre>
\ <lf></lf>	<pre>pass the PCRE_NEWLINE_LF option to pcre_exec() or pcre_</pre>
\ <crlf></crlf>	<pre>pass the PCRE_NEWLINE_CRLF option to pcre_exec() or pcr</pre>
<pre>\<anycrlf></anycrlf></pre>	<pre>pass the PCRE_NEWLINE_ANYCRLF option to pcre_exec() or</pre>
\ <any></any>	pass the PCRE_NEWLINE_ANY option to pcre_exec() or pcre

The escapes that specify line ending sequences are literal strings, exactly as shown. No more than one newline setting should be present in any data line.

A backslash followed by anything else just escapes the anything else. If the very last character is a backslash, it is ignored. This gives a way of passing an empty line as data, since a real empty line terminates the data input.

If \M is present, **pcretest** calls **pcre_exec()** several times, with different values in the *match_limit* and *match_limit_recursion* fields of the **pcre_extra** data structure, until it finds the minimum numbers for each parameter that allow **pcre_exec()** to complete. The *match_limit* number is a measure of the amount of backtracking that takes place, and checking it out can be instructive. For most simple matches, the number is quite small, but for patterns with very large numbers of matching possibilities, it can become large very quickly with increasing length of subject string. The *match_limit_recursion* number is a measure of how much stack (or, if PCRE is compiled with NO_RECURSE, how much heap) memory is needed to complete the match attempt.

When \O is used, the value specified may be higher or lower than the size set by the **-O** command line option (or defaulted to 45); \O applies only to the call of **pcre_exec()** for the line in which it appears.

If the **/P** modifier was present on the pattern, causing the POSIX wrapper API to be used, the only option-setting sequences that have any effect are \B and \Z, causing REG_NOTBOL and REG_NOTEOL, respectively, to be passed to **regexec()**.

The use of $x{hh...}$ to represent UTF-8 characters is not dependent on the use of the /**8** modifier on the pattern. It is recognized always. There may be any number of hexadecimal digits inside the braces. The result is from one to six bytes, encoded according to the UTF-8 rules.

THE ALTERNATIVE MATCHING FUNCTION

By default, **pcretest** uses the standard PCRE matching function, **pcre_exec()** to match each data line. From release 6.0, PCRE supports an alternative matching function, **pcre_dfa_test()**, which operates in a different way, and has some restrictions. The differences between the two functions are described in the **pcrematching** documentation.

If a data line contains the \D escape sequence, or if the command line contains the **-dfa** option, the alternative matching function is called. This function finds all possible matches at a given point. If, however, the \F escape sequence is present in the data line, it stops after the first match is found. This is always the shortest possible match.

DEFAULT OUTPUT FROM PCRETEST

This section describes the output when the normal matching function, **pcre_exec()**, is being used.

When a match succeeds, pcretest outputs the list of captured substrings that **pcre_exec()** returns, starting with number 0 for the string that matched the whole pattern. Otherwise, it outputs "No match" or "Partial match" when **pcre_exec()** returns PCRE_ERROR_NOMATCH or PCRE_ERROR_PARTIAL, respectively, and otherwise the PCRE negative error number. Here is an example of an interactive **pcretest** run.

\$ pcretest
PCRE version 7.0 30-Nov-2006

```
re> /^abc(\d+)/
data> abc123
0: abc123
1: 123
data> xyz
No match
```

If the strings contain any non-printing characters, they are output as 0x escapes, or as $x\{...\}$ escapes if the **/8** modifier was present on the pattern. See below for the definition of non-printing characters. If the pattern has the **/+** modifier, the output for substring 0 is followed by the the rest of the subject string, identified by "0+" like this:

```
re> /cat/+
data> cataract
0: cat
0+ aract
```

If the pattern has the /g or /G modifier, the results of successive matching attempts are output in sequence, like this:

```
re> /\Bi(\w\w)/g
data> Mississippi
0: iss
1: ss
0: iss
1: ss
0: ipp
1: pp
```

"No match" is output only if the first match attempt fails.

If any of the sequences \C , \G , or \L are present in a data line that is successfully matched, the substrings extracted by the convenience functions are output with C, G, or L after the string number instead of a colon. This is in addition to the normal full list. The string length (that is, the return from the extraction function) is given in parentheses after each string for \C and \G .

Note that whereas patterns can be continued over several lines (a plain ">" prompt is used for continuations), data lines may not. However newlines can be included in data by means of the \n escape (or \r, \r\n, etc., depending on the newline sequence setting).

OUTPUT FROM THE ALTERNATIVE MATCHING FUNCTION

When the alternative matching function, **pcre_dfa_exec()**, is used (by means of the \D escape sequence or the **-dfa** command line option), the output consists of a list of all the matches that start at the first point in the subject where there is at least one match. For example:

```
re> /(tang|tangerine|tan)/
data> yellow tangerine\D
0: tangerine
1: tang
2: tan
```

(Using the normal matching function on this data finds only "tang".) The longest matching string is always given first (and numbered zero).

If **/g** is present on the pattern, the search for further matches resumes at the end of the longest match. For example:

```
re> /(tang|tangerine|tan)/g
data> yellow tangerine and tangy sultana\D
0: tangerine
1: tang
2: tan
0: tang
1: tan
0: tan
```

Since the matching function does not support substring capture, the escape sequences that are concerned with captured substrings are not relevant.

RESTARTING AFTER A PARTIAL MATCH

When the alternative matching function has given the PCRE_ERROR_PARTIAL return, indicating that the subject partially matched the pattern, you can restart the match with additional subject data by means of the \R escape sequence. For example:

```
re> /^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d
data> 23ja\P\D
Partial match: 23ja
data> n05\R\D
0: n05
```

For further information about partial matching, see the **<u>pcrepartial</u>** documentation.

CALLOUTS

If the pattern contains any callout requests, **pcretest**'s callout function is called during matching. This works with both matching functions. By default, the called function displays the callout number, the start and current positions in the text at the callout time, and the next pattern item to be tested. For example, the output

--->pqrabcdef 0 ^ ^ \d

indicates that callout number 0 occurred for a match attempt starting at the fourth character of the subject string, when the pointer was at the seventh character of the data, and when the next pattern item was \d. Just one circumflex is output if the start and current positions are the same.

Callouts numbered 255 are assumed to be automatic callouts, inserted as a result of the /**C** pattern modifier. In this case, instead of showing the callout number, the offset in the pattern, preceded by a plus, is output. For example:

```
re> /\d?[A-E]\*/C
data> E*
--->E*
+0 ^ \d?
+3 ^ [A-E]
+8 ^^ \*
+10 ^ ^
0: E*
```

The callout function in **pcretest** returns zero (carry on matching) by default, but you can use a \C item in a data line (as described above) to change this.

Inserting callouts can be helpful when using **pcretest** to check complicated regular expressions. For further information about callouts, see the **pcrecallout** documentation.

NON-PRINTING CHARACTERS

When **pcretest** is outputting text in the compiled version of a pattern, bytes other than 32-126 are always treated as non-printing characters are are therefore shown as hex escapes.

When **pcretest** is outputting text that is a matched part of a subject string, it behaves in the same way, unless a different locale has been set for the pattern (using the /L modifier). In this case, the **isprint()** function to distinguish printing and non-printing characters.

SAVING AND RELOADING COMPILED PATTERNS

The facilities described in this section are not available when the POSIX inteface to PCRE is being used, that is, when the $/\mathbf{P}$ pattern modifier is specified.

When the POSIX interface is not in use, you can cause **pcretest** to write a compiled pattern to a file, by following the modifiers with > and a file name. For example:

/pattern/im >/some/file

See the **<u>pcreprecompile</u>** documentation for a discussion about saving and reusing compiled patterns.

The data that is written is binary. The first eight bytes are the length of the compiled pattern data followed by the length of the optional study data, each written as four bytes in big-endian order (most significant byte first). If there is no study data (either the pattern was not studied, or studying did not return any data), the second length is zero. The lengths are followed by an exact copy of the compiled pattern. If there is additional study data, this follows immediately after the compiled pattern. After writing the file, **pcretest** expects to read a new pattern.

A saved pattern can be reloaded into **pcretest** by specifing < and a file name instead of a pattern. The name of the file must not contain a < character, as otherwise **pcretest** will interpret the line as a pattern delimited by < characters. For example:

```
re> </some/file
Compiled regex loaded from /some/file
No study data</pre>
```

When the pattern has been loaded, **pcretest** proceeds to read data lines in the usual way.

You can copy a file written by **pcretest** to a different host and reload it there, even if the new host has opposite endianness to the one on which the pattern was compiled. For example, you can compile on an i86 machine and run on a SPARC machine.

File names for saving and reloading can be absolute or relative, but note that the shell facility of expanding a file name that starts with a tilde (~) is not available.

The ability to save and reload files in **pcretest** is intended for testing and experimentation. It is not intended for production use because only a single pattern can be written to a file. Furthermore, there is no facility for supplying custom character tables for use with a reloaded pattern. If the original pattern was compiled with custom tables, an attempt to match a subject string using a reloaded pattern is likely to cause **pcretest** to crash. Finally, if you attempt to load a file that is not in the correct format, the result is undefined.

SEE ALSO

pcre(3), pcreapi(3), pcrecallout(3), pcrematching(3), pcrepartial(d), pcrepattern(3), pcreprecompile(3).

AUTHOR

Philip Hazel University Computing Service Cambridge CB2 3QH, England.

REVISION

Last updated: 24 April 2007 Copyright © 1997-2007 University of Cambridge.

Return to the <u>PCRE index page</u>.