NAME

PCRE - Perl-compatible regular expressions

#include <pcre.h>

PCRE NATIVE API BASIC FUNCTIONS

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*);*

void pcre_free_study(pcre_extra *****extra*);*

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***);**

PCRE NATIVE API STRING EXTRACTION FUNCTIONS

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_);**

PCRE NATIVE API AUXILIARY FUNCTIONS

int pcre_jit_exec(const pcre ***_code_, const pcre_extra ***_extra_**,**
const char ***_subject_, int** _length_**, int** _startoffset_**,**
int _options_**, int ***_ovector_**, int** _ovecsize_**,**
pcre_jit_stack ***_jstack_);**

pcre_jit_stack *pcre_jit_stack_alloc(int _startsize_**, int** _maxsize_**);**

void pcre_jit_stack_free(pcre_jit_stack ***_stack_);**

void pcre_assign_jit_stack(pcre_extra ***_extra_,**
pcre_jit_callback _callback_**, void ***_data_**);**

const unsigned char *pcre_maketables(void);

int pcre_fullinfo(const pcre ***_code_, const pcre_extra ***_extra_**,**
int _what_**, void ***_where_**);**

int pcre_refcount(pcre ***_code_, int** _adjust_**);**

int pcre_config(int *what***, void ****where***);**

const char *pcre_version(void);

int pcre_pattern_to_host_byte_order(pcre ****code*,**
pcre_extra ****extra*, const unsigned char ****tables***);**

PCRE NATIVE API INDIRECTED FUNCTIONS

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 *);

int (*pcre_stack_guard)(void);

PCRE 8-BIT, 16-BIT, AND 32-BIT LIBRARIES

As well as support for 8-bit character strings, PCRE also supports 16-bit strings (from release 8.30) and 32-bit strings (from release 8.32), by means of two additional libraries. They can be built as well as, or instead of, the 8-bit library. To avoid too much complication, this document describes the 8-bit versions of the functions, with only occasional references to the 16-bit and 32-bit libraries.

The 16-bit and 32-bit functions operate in the same way as their 8-bit counterparts; they just use different data types for their arguments and results, and their names start with **pcre16_** or **pcre32_** instead of **pcre_**. For every option that has UTF8 in its name (for example, PCRE_UTF8), there are corresponding 16-bit and 32-bit names with UTF8 replaced by UTF16 or UTF32, respectively. This facility is in fact just cosmetic; the 16-bit and 32-bit option names define the same bit values.

References to bytes and UTF-8 in this document should be read as references to 16-bit data units and UTF-16 when using the 16-bit library, or 32-bit data units and UTF-32 when using the 32-bit library, unless specified otherwise. More details of the specific differences for the 16-bit and 32-bit libraries are given in the **pcre16** and **pcre32** pages.

PCRE API OVERVIEW

PCRE has its own native API, which is described in this document. There are also some wrapper functions (for the 8-bit library only) that correspond to the POSIX regular expression API, but they do

not give access to all the functionality. They are described in the **pcreposix** documentation. Both of these APIs define a set of C function calls. A C++ wrapper (again for the 8-bit library only) is also 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-like systems the (8-bit) 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.

In a Windows environment, if you want to statically link an application program against a non-dll **pcre.a** file, you must define PCRE_STATIC before including **pcre.h** or **pcrecpp.h**, because otherwise the **pcre_malloc()** and **pcre_free()** exported functions will be declared **__declspec(dllimport)**, with unwanted results.

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 PCRE source distribution. A listing of this program is given in the **pcredemo** documentation, and the **pcresample** documentation describes how to compile and run it.

Just-in-time compiler support is an optional feature of PCRE that can be built in appropriate hardware environments. It greatly speeds up the matching performance of many patterns. Simple programs can easily request that it be used if available, by setting an option that is ignored when it is not relevant. More complicated programs might need to make use of the functions **pcre_jit_stack_alloc()**, **pcre_jit_stack_free()**, and **pcre_assign_jit_stack()** in order to control the JIT code’s memory usage.

From release 8.32 there is also a direct interface for JIT execution, which gives improved performance. The JIT-specific functions are discussed in the **pcrejit** documentation.

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 (unless there are lookbehind assertions). 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. 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 always allocate 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.

The global variable **pcre_stack_guard** initially contains NULL. It can be set by the caller to a function

that is called by PCRE whenever it starts to compile a parenthesized part of a pattern. When parentheses are nested, PCRE uses recursive function calls, which use up the system stack. This function is provided so that applications with restricted stacks can force a compilation error if the stack runs out. The function should return zero if all is well, or non-zero to force an error.

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 (form feed, 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.

At compile time, the newline convention can be specified by the *options* argument of **pcre_compile()**, or it can be specified by special text at the start of the pattern itself; this overrides any other settings. See the **pcrepattern** page for details of the special character sequences.

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. There is more detail about this in the section on **pcre_exec()** options below.

The choice of newline convention does not affect the interpretation of the \n or \r escape sequences, nor does it affect what \R matches, which is controlled in a similar way, but by separate options.

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 and stack-checking functions pointed to by **pcre_callout** and **pcre_stack_guard**, 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.

If the just-in-time optimization feature is being used, it needs separate memory stack areas for each

thread. See the **pcrejit** documentation for more details.

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, which includes a description of the **pcre_pattern_to_host_byte_order()** function. 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 returned value is zero on success, or the negative error code PCRE_ERROR_BADOPTION if the value in the first argument is not recognized. 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. This value should normally be given to the 8-bit version of this function, **pcre_config()**. If it is given to the 16-bit or 32-bit version of this function, the result is PCRE_ERROR_BADOPTION.

PCRE_CONFIG_UTF16

The output is an integer that is set to one if UTF-16 support is available; otherwise it is set to zero. This value should normally be given to the 16-bit version of this function, **pcre16_config()**. If it is given to the 8-bit or 32-bit version of this function, the result is PCRE_ERROR_BADOPTION.

PCRE_CONFIG_UTF32

The output is an integer that is set to one if UTF-32 support is available; otherwise it is set to zero. This value should normally be given to the 32-bit version of this function, **pcre32_config()**. If it is given to the 8-bit or 16-bit version of this function, the result is PCRE_ERROR_BADOPTION.

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_JIT

The output is an integer that is set to one if support for just-in-time compiling is available; otherwise it is set to zero.

PCRE_CONFIG_JITTARGET

The output is a pointer to a zero-terminated "const char *" string. If JIT support is available, the string contains the name of the architecture for which the JIT compiler is configured, for example "x86 32bit (little endian + unaligned)". If JIT support is not available, the result is NULL.

PCRE_CONFIG_NEWLINE

The output is an integer whose value specifies the default character sequence that is recognized as meaning "newline". The values that are supported in ASCII/Unicode environments are: 10 for LF, 13 for CR, 3338 for CRLF, -2 for ANYCRLF, and -1 for ANY. In EBCDIC environments, CR, ANYCRLF, and ANY yield the same values. However, the value for LF is normally 21, though some EBCDIC environments use 37. The corresponding values for CRLF are 3349 and 3365. The default should normally correspond to the standard sequence for your operating system.

PCRE_CONFIG_BSR

The output is an integer whose value indicates what character sequences the \R escape sequence matches by default. A value of 0 means that \R matches any Unicode line ending sequence; a value of 1 means that \R matches only CR, LF, or CRLF. The default can be overridden when a pattern is compiled or matched.

PCRE_CONFIG_LINK_SIZE

The output is an integer that contains the number of bytes used for internal linkage in compiled regular expressions. For the 8-bit library, the value can be 2, 3, or 4. For the 16-bit library, the value is either 2 or 4 and is still a number of bytes. For the 32-bit library, the value is either 2 or 4 and is still a number of bytes. 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. Larger values allow larger regular expressions to be compiled, at the expense of slower matching.

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_PARENS_LIMIT

The output is a long integer that gives the maximum depth of nesting of parentheses (of any kind) in a pattern. This limit is imposed to cap the amount of system stack used when a pattern is compiled. It is specified when PCRE is built; the default is 250. This limit does not take into account the stack that may already be used by the calling application. For finer control over compilation stack usage, you can set a pointer to an external checking function in **pcre_stack_guard**.

PCRE_CONFIG_MATCH_LIMIT

The output is a long 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 a long 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. To avoid too much repetition, we refer just to **pcre_compile()** below, but the information applies equally to **pcre_compile2()**.

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, but some others as well) can also be set and unset from within the pattern (see the detailed description in the **pcrepattern** documentation). For those options that can be different in different parts of the pattern, the contents of the *options* argument specifies their settings at the start of compilation and execution. The PCRE_ANCHORED, PCRE_BSR_*xxx*, PCRE_NEWLINE_*xxx*, PCRE_NO_UTF8_CHECK, and PCRE_NO_START_OPTIMIZE 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. Normally, the offset from the start of the pattern to the data unit that was being processed when 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). However, for an invalid UTF-8 or UTF-16 string, the offset is that of the first data unit of the failing character.

Some errors are not detected until the whole pattern has been scanned; in these cases, the offset passed back is the length of the pattern. Note that the offset is in data units, not characters, even in a UTF mode. It may sometimes point into the middle of a UTF-8 or UTF-16 character.

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()** and **pcre_dfa_exec()** when the pattern is matched. For more discussion, see the section on locale support below.

This code fragment shows a typical straightforward call to **pcre_compile()**:

```
pcre *re;
const char *error;
int erroffset;
re = pcre_compile(
  "^A.*Z",          /* the pattern */
  0,                /* default options */
  &error,           /* for error message */
  &erroffset,       /* for error offset */
  NULL);            /* use default character tables */
```

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_BSR_ANYCRLF
PCRE_BSR_UNICODE

These options (which are mutually exclusive) control what the \R escape sequence matches. The choice is either to match only CR, LF, or CRLF, or to match any Unicode newline sequence. The default is specified when PCRE is built. It can be overridden from within the pattern, or by setting an option when a compiled pattern is matched.

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 metacharacter in the pattern matches a character of any value, including one that indicates a newline. However, it only ever matches one character, even if newlines are coded as CRLF. Without this option, 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, most white space characters in the pattern are totally ignored except when escaped or inside a character class. However, white space is not allowed within sequences such as (?> that introduce various parenthesized subpatterns, nor within a numerical quantifier such as {1,3}. However, ignorable white space is permitted between an item and a following quantifier and between a quantifier and a following + that indicates possessiveness.

White space did not used to include the VT character (code 11), because Perl did not treat this character as white space. However, Perl changed at release 5.18, so PCRE followed at release 8.34, and VT is

now treated as white space.

PCRE_EXTENDED also causes characters between an unescaped # outside a character class and the next newline, inclusive, to be ignored. PCRE_EXTENDED is equivalent to Perl's /x option, and it can be changed within a pattern by a (?x) option setting.

Which characters are interpreted as newlines is controlled by the options passed to **pcre_compile()** or by a special sequence at the start of the pattern, as described in the section entitled "Newline conventions" in the **pcrepattern** documentation. Note that the end of this type of comment is a literal newline sequence in the pattern; escape sequences that happen to represent a newline do not count.

This option makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. White space characters may never appear within special character sequences in a pattern, for example within the sequence (?(that 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 an error for this, by running it with the -w option.) 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_JAVASCRIPT_COMPAT

If this option is set, PCRE's behaviour is changed in some ways so that it is compatible with JavaScript rather than Perl. The changes are as follows:

(1) A lone closing square bracket in a pattern causes a compile-time error, because this is illegal in JavaScript (by default it is treated as a data character). Thus, the pattern AB]CD becomes illegal when this option is set.

(2) At run time, a back reference to an unset subpattern group matches an empty string (by default this

causes the current matching alternative to fail). A pattern such as (\1)(a) succeeds when this option is set (assuming it can find an "a" in the subject), whereas it fails by default, for Perl compatibility.

(3) \U matches an upper case "U" character; by default \U causes a compile time error (Perl uses \U to upper case subsequent characters).

(4) \u matches a lower case "u" character unless it is followed by four hexadecimal digits, in which case the hexadecimal number defines the code point to match. By default, \u causes a compile time error (Perl uses it to upper case the following character).

(5) \x matches a lower case "x" character unless it is followed by two hexadecimal digits, in which case the hexadecimal number defines the code point to match. By default, as in Perl, a hexadecimal number is always expected after \x, but it may have zero, one, or two digits (so, for example, \xz matches a binary zero character followed by z).

PCRE_MULTILINE

By default, for the purposes of matching "start of line" and "end of line", 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, and the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline (except when PCRE_DOLLAR_ENDONLY is set). Note, however, that unless PCRE_DOTALL is set, the "any character" metacharacter (.) does not match at a newline. This behaviour (for ^, $, and dot) 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_NEVER_UTF

This option locks out interpretation of the pattern as UTF-8 (or UTF-16 or UTF-32 in the 16-bit and 32-bit libraries). In particular, it prevents the creator of the pattern from switching to UTF interpretation by starting the pattern with (*UTF). This may be useful in applications that process patterns from external sources. The combination of PCRE_UTF8 and PCRE_NEVER_UTF also causes an error.

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 two-character 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.

In an ASCII/Unicode environment, the Unicode newline sequences are the three just mentioned, plus the single characters VT (vertical tab, U+000B), FF (form feed, U+000C), NEL (next line, U+0085), LS (line separator, U+2028), and PS (paragraph separator, U+2029). For the 8-bit library, the last two are recognized only in UTF-8 mode.

When PCRE is compiled to run in an EBCDIC (mainframe) environment, the code for CR is 0x0d, the same as ASCII. However, the character code for LF is normally 0x15, though in some EBCDIC environments 0x25 is used. Whichever of these is not LF is made to correspond to Unicode's NEL character. EBCDIC codes are all less than 256. For more details, see the **pcrebuild** documentation.

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 in a pattern is specially recognized when compiling is when PCRE_EXTENDED is set. CR and LF are white space characters, and so are ignored in this mode. Also, an unescaped # outside a character class indicates a comment that lasts until after the next line break sequence. In other circumstances, line break sequences in patterns are treated as literal data.

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_NO_AUTO_POSSESS

If this option is set, it disables "auto-possessification". This is an optimization that, for example, turns a+b into a++b in order to avoid backtracks into a+ that can never be successful. However, if callouts are in use, auto-possessification means that some of them are never taken. You can set this option if you want the matching functions to do a full unoptimized search and run all the callouts, but it is mainly provided for testing purposes.

PCRE_NO_START_OPTIMIZE

This is an option that acts at matching time; that is, it is really an option for **pcre_exec()** or **pcre_dfa_exec()**. If it is set at compile time, it is remembered with the compiled pattern and assumed at matching time. This is necessary if you want to use JIT execution, because the JIT compiler needs to know whether or not this option is set. For details see the discussion of PCRE_NO_START_OPTIMIZE below.

PCRE_UCP

This option changes the way PCRE processes \B, \b, \D, \d, \S, \s, \W, \w, and some of the POSIX character classes. By default, only ASCII characters are recognized, but if PCRE_UCP is set, Unicode properties are used instead to classify characters. More details are given in the section on generic character types in the **pcrepattern** page. If you set PCRE_UCP, matching one of the items it affects takes much longer. The option is available only if PCRE has been compiled with Unicode property support.

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 strings. However, it is available only when PCRE is built to include UTF 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 **pcreunicode** page.

PCRE_NO_UTF8_CHECK

When PCRE_UTF8 is set, the validity of the pattern as a UTF-8 string is automatically checked. There is a discussion about the validity of UTF-8 strings in the **pcreunicode** page. If an invalid UTF-8 sequence 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 or loop. Note that this option can also be passed to **pcre_exec()** and **pcre_dfa_exec()**, to suppress the validity checking of subject strings only. If the same string is being matched many times, the option can be safely set for the second and subsequent matchings to improve performance.

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. Note that error messages are always 8-bit ASCII strings, even in 16-bit or 32-bit mode. 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
5 number too big in {} quantifier
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 (? or (?-
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 is 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 compiled without UTF support
33 [this code is not in use]
34 character value in \x{} or \o{} is too large
35 invalid condition (?(0)
36 \C not allowed in lookbehind assertion
37 PCRE does not support \L, \l, \N{name}, \U, or \u
38 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 (specifically UTF-8)
45 support for \P, \p, and \X has not been compiled
46 malformed \P or \p sequence
47 unknown property name after \P or \p
48 subpattern name is too long (maximum 32 characters)
49 too many named subpatterns (maximum 10000)
50 [this code is not in use]
51 octal value is greater than \377 in 8-bit non-UTF-8 mode
52 internal error: overran compiling workspace
53 internal error: previously-checked referenced subpattern
not found
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, angle-bracketed, or quoted
name/number or by a plain number
58 a numbered reference must not be zero
59 an argument is not allowed for (*ACCEPT), (*FAIL), or (*COMMIT)

60 (*VERB) not recognized or malformed
61 number is too big
62 subpattern name expected
63 digit expected after (?+
64] is an invalid data character in JavaScript compatibility mode
65 different names for subpatterns of the same number are
not allowed
66 (*MARK) must have an argument
67 this version of PCRE is not compiled with Unicode property
support
68 \c must be followed by an ASCII character
69 \k is not followed by a braced, angle-bracketed, or quoted name
70 internal error: unknown opcode in find_fixedlength()
71 \N is not supported in a class
72 too many forward references
73 disallowed Unicode code point (>= 0xd800 && <= 0xdfff)
74 invalid UTF-16 string (specifically UTF-16)
75 name is too long in (*MARK), (*PRUNE), (*SKIP), or (*THEN)
76 character value in \u.... sequence is too large
77 invalid UTF-32 string (specifically UTF-32)
78 setting UTF is disabled by the application
79 non-hex character in \x{} (closing brace missing?)
80 non-octal character in \o{} (closing brace missing?)
81 missing opening brace after \o
82 parentheses are too deeply nested
83 invalid range in character class
84 group name must start with a non-digit
85 parentheses are too deeply nested (stack check)

The numbers 32 and 10000 in errors 48 and 49 are defaults; different values may be used if the limits were changed when PCRE was built.

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()** or **pcre_dfa_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 below in the section on matching a pattern.

If studying the pattern does not produce any useful information, **pcre_study()** returns NULL by default. In that circumstance, if the calling program wants to pass any of the other fields to **pcre_exec()** or **pcre_dfa_exec()**, it must set up its own **pcre_extra** block. However, if **pcre_study()** is called with the PCRE_STUDY_EXTRA_NEEDED option, it returns a **pcre_extra** block even if studying did not find any additional information. It may still return NULL, however, if an error occurs in **pcre_study()**.

The second argument of **pcre_study()** contains option bits. There are three further options in addition to PCRE_STUDY_EXTRA_NEEDED:

PCRE_STUDY_JIT_COMPILE
PCRE_STUDY_JIT_PARTIAL_HARD_COMPILE
PCRE_STUDY_JIT_PARTIAL_SOFT_COMPILE

If any of these are set, and the just-in-time compiler is available, the pattern is further compiled into machine code that executes much faster than the **pcre_exec()** interpretive matching function. If the just-in-time compiler is not available, these options are ignored. All undefined bits in the *options* argument must be zero.

JIT compilation is a heavyweight optimization. It can take some time for patterns to be analyzed, and for one-off matches and simple patterns the benefit of faster execution might be offset by a much slower study time. Not all patterns can be optimized by the JIT compiler. For those that cannot be handled, matching automatically falls back to the **pcre_exec()** interpreter. For more details, see the **pcrejit** documentation.

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.

When you are finished with a pattern, you can free the memory used for the study data by calling **pcre_free_study()**. This function was added to the API for release 8.20. For earlier versions, the memory could be freed with **pcre_free()**, just like the pattern itself. This will still work in cases where JIT optimization is not used, but it is advisable to change to the new function when convenient.

This is a typical way in which **pcre_study**() is used (except that in a real application there should be tests for errors):

```
int rc;
pcre *re;
pcre_extra *sd;
re = pcre_compile("pattern", 0, &error, &erroroffset, NULL);
sd = pcre_study(
  re,           /* result of pcre_compile() */
  0,            /* no options */
  &error);      /* set to NULL or points to a message */
rc = pcre_exec(   /* see below for details of pcre_exec() options */
  re, sd, "subject", 7, 0, 0, ovector, 30);
...
pcre_free_study(sd);
pcre_free(re);
```

Studying a pattern does two things: first, a lower bound for the length of subject string that is needed to match the pattern is computed. This does not mean that there are any strings of that length that match, but it does guarantee that no shorter strings match. The value is used to avoid wasting time by trying to match strings that are shorter than the lower bound. You can find out the value in a calling program via the **pcre_fullinfo()** function.

Studying a pattern is also useful for non-anchored patterns that do not have a single fixed starting character. A bitmap of possible starting bytes is created. This speeds up finding a position in the subject at which to start matching. (In 16-bit mode, the bitmap is used for 16-bit values less than 256. In 32-bit mode, the bitmap is used for 32-bit values less than 256.)

These two optimizations apply to both **pcre_exec()** and **pcre_dfa_exec()**, and the information is also used by the JIT compiler. The optimizations can be disabled by setting the PCRE_NO_START_OPTIMIZE option. You might want to do this if your pattern contains callouts or (*MARK) and you want to make use of these facilities in cases where matching fails.

PCRE_NO_START_OPTIMIZE can be specified at either compile time or execution time. However, if PCRE_NO_START_OPTIMIZE is passed to **pcre_exec()**, (that is, after any JIT compilation has happened) JIT execution is disabled. For JIT execution to work with PCRE_NO_START_OPTIMIZE, the option must be set at compile time.

There is a longer discussion of PCRE_NO_START_OPTIMIZE below.

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 code point. When running in UTF-8 mode, or in the 16- or 32-bit libraries, this applies only to characters with code points less than 256. By default, higher-valued code points never match escapes such as \w or \d. However, if PCRE is built with Unicode property support, all characters can be tested with \p and \P, or, alternatively, the PCRE_UCP option can be set when a pattern is compiled; this causes \w and friends to use Unicode property support instead of the built-in tables.

The use of locales with Unicode is discouraged. If you are handling characters with code points greater than 128, you should either use Unicode support, 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()** 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 also by **pcre_exec()** and **pcre_dfa_exec()**. Thus, for any single pattern, compilation, studying and matching all happen in the same locale, but different patterns can be

processed in different locales.

It is possible to pass a table pointer or NULL (indicating the use of the internal tables) to **pcre_exec()** or **pcre_dfa_exec()** (see the discussion below in the section on matching a pattern). This facility is provided for use with pre-compiled patterns that have been saved and reloaded. Character tables are not saved with patterns, so if a non-standard table was used at compile time, it must be provided again when the reloaded pattern is matched. Attempting to use this facility to match a pattern in a different locale from the one in which it was compiled is likely to lead to anomalous (usually incorrect) results.

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 **pcre_info()** function, which was removed from the library at version 8.30, after more than 10 years of obsolescence.

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 code was NULL
                         the argument where was NULL
PCRE_ERROR_BADMAGIC      the "magic number" was not found
PCRE_ERROR_BADENDIANNESS the pattern was compiled with different
                         endianness
PCRE_ERROR_BADOPTION     the value of what was invalid
PCRE_ERROR_UNSET         the requested field is not set
```

The "magic number" is placed at the start of each compiled pattern as a simple check against passing an arbitrary memory pointer. The endianness error can occur if a compiled pattern is saved and reloaded on a different host. Here is a typical call of **pcre_fullinfo()**, to obtain the length of the compiled pattern:

```
int rc;
size_t length;
rc = pcre_fullinfo(
  re,               /* result of pcre_compile() */
  sd,               /* result of pcre_study(), or NULL */
  PCRE_INFO_SIZE,   /* what is required */
```

```
  &length);       /* where to put the data */
```

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 (deprecated)

Return information about the first data unit of any matched string, for a non-anchored pattern. The name of this option refers to the 8-bit library, where data units are bytes. The fourth argument should point to an **int** variable. Negative values are used for special cases. However, this means that when the 32-bit library is in non-UTF-32 mode, the full 32-bit range of characters cannot be returned. For this reason, this value is deprecated; use PCRE_INFO_FIRSTCHARACTERFLAGS and PCRE_INFO_FIRSTCHARACTER instead.

If there is a fixed first value, for example, the letter "c" from a pattern such as (cat|cow|coyote), its value is returned. In the 8-bit library, the value is always less than 256. In the 16-bit library the value can be up to 0xffff. In the 32-bit library the value can be up to 0x10ffff.

If there is no fixed first value, and 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_FIRSTCHARACTER

Return the value of the first data unit (non-UTF character) of any matched string in the situation where PCRE_INFO_FIRSTCHARACTERFLAGS returns 1; otherwise return 0. The fourth argument should point to a **uint_t** variable.

In the 8-bit library, the value is always less than 256. In the 16-bit library the value can be up to 0xffff. In the 32-bit library in UTF-32 mode the value can be up to 0x10ffff, and up to 0xffffffff when not using UTF-32 mode.

PCRE_INFO_FIRSTCHARACTERFLAGS

Return information about the first data unit of any matched string, for a non-anchored pattern. The fourth argument should point to an **int** variable.

If there is a fixed first value, for example, the letter "c" from a pattern such as (cat|cow|coyote), 1 is returned, and the character value can be retrieved using PCRE_INFO_FIRSTCHARACTER. If there is no fixed first value, and 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),

2 is returned, indicating that the pattern matches only at the start of a subject string or after any newline within the string. Otherwise 0 is returned. For anchored patterns, 0 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 values for the first data unit 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_HASCRORLF

Return 1 if the pattern contains any explicit matches for CR or LF characters, otherwise 0. The fourth argument should point to an **int** variable. An explicit match is either a literal CR or LF character, or \r

or \n.

PCRE_INFO_JCHANGED

Return 1 if the (?J) or (?-J) option setting is used in the pattern, otherwise 0. The fourth argument should point to an **int** variable. (?J) and (?-J) set and unset the local PCRE_DUPNAMES option, respectively.

PCRE_INFO_JIT

Return 1 if the pattern was studied with one of the JIT options, and just-in-time compiling was successful. The fourth argument should point to an **int** variable. A return value of 0 means that JIT support is not available in this version of PCRE, or that the pattern was not studied with a JIT option, or that the JIT compiler could not handle this particular pattern. See the **pcrejit** documentation for details of what can and cannot be handled.

PCRE_INFO_JITSIZE

If the pattern was successfully studied with a JIT option, return the size of the JIT compiled code, otherwise return zero. The fourth argument should point to a **size_t** variable.

PCRE_INFO_LASTLITERAL

Return the value of the rightmost literal data unit that must exist in any matched string, other than at its start, if such a value has been recorded. The fourth argument should point to an **int** variable. If there is no such value, -1 is returned. For anchored patterns, a last literal value 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.

Since for the 32-bit library using the non-UTF-32 mode, this function is unable to return the full 32-bit range of characters, this value is deprecated; instead the PCRE_INFO_REQUIREDCHARFLAGS and PCRE_INFO_REQUIREDCHAR values should be used.

PCRE_INFO_MATCH_EMPTY

Return 1 if the pattern can match an empty string, otherwise 0. The fourth argument should point to an **int** variable.

PCRE_INFO_MATCHLIMIT

If the pattern set a match limit by including an item of the form (*LIMIT_MATCH=nnnn) at the start, the value is returned. The fourth argument should point to an unsigned 32-bit integer. If no such value has been set, the call to **pcre_fullinfo()** returns the error PCRE_ERROR_UNSET.

PCRE_INFO_MAXLOOKBEHIND

Return the number of characters (NB not data units) in the longest lookbehind assertion in the pattern. This information is useful when doing multi-segment matching using the partial matching facilities. Note that the simple assertions \b and \B require a one-character lookbehind. \A also registers a one-character lookbehind, though it does not actually inspect the previous character. This is to ensure that at least one character from the old segment is retained when a new segment is processed. Otherwise, if there are no lookbehinds in the pattern, \A might match incorrectly at the start of a new segment.

PCRE_INFO_MINLENGTH

If the pattern was studied and a minimum length for matching subject strings was computed, its value is returned. Otherwise the returned value is -1. The value is a number of characters, which in UTF mode may be different from the number of data units. The fourth argument should point to an **int** variable. A non-negative value is a lower bound to the length of any matching string. There may not be any strings of that length that do actually match, but every string that does match is at least that long.

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. This is a pointer to **char** in the 8-bit library, where the first two bytes of each entry are the number of the capturing parenthesis, most significant byte first. In the 16-bit library, the pointer points to 16-bit data units, the first of which contains the parenthesis number. In the 32-bit library, the pointer points to 32-bit data units, the first of which contains the parenthesis number. The rest of the entry is the corresponding name, zero terminated.

The names are in alphabetical order. If (?| is used to create multiple groups with the same number, as described in the section on duplicate subpattern numbers in the **pcrepattern** page, the groups may be given the same name, but there is only one entry in the table. Different names for groups of the same number are not permitted. Duplicate names for subpatterns with different numbers are permitted, but only if PCRE_DUPNAMES is set. They appear in the table in the order in which they were found in the pattern. In the absence of (?| this is the order of increasing number; when (?| is used this is not necessarily the case because later subpatterns may have lower numbers.

As a simple example of the name/number table, consider the following pattern after compilation by the 8-bit library (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  a  t  e  00 ??
00 05 d  a  y  00 ?? ??
00 04 m  o  n  t  h  00
00 02 y  e  a  r  00 ??
```

When writing code to extract data from named subpatterns using the name-to-number map, remember that the length of the entries is likely to be different for each compiled pattern.

PCRE_INFO_OKPARTIAL

Return 1 if the pattern can be used for partial matching with **pcre_exec()**, otherwise 0. The fourth argument should point to an **int** variable. From release 8.00, this always returns 1, because the restrictions that previously applied to partial matching have been lifted. The **pcrepartial** documentation gives details of partial matching.

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 at the start of the pattern itself. In other words, they are the options that will be in force when matching starts. For example, if the pattern /(?im)abc(?-i)d/ is compiled with the PCRE_EXTENDED option, the result is PCRE_CASELESS, PCRE_MULTILINE,

and PCRE_EXTENDED.

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
\A    always
\G    always
.*    if PCRE_DOTALL is set and there are no back
        references to the subpattern in which .* appears
```

For such patterns, the PCRE_ANCHORED bit is set in the options returned by **pcre_fullinfo()**.

PCRE_INFO_RECURSIONLIMIT

If the pattern set a recursion limit by including an item of the form (*LIMIT_RECURSION=nnnn) at the start, the value is returned. The fourth argument should point to an unsigned 32-bit integer. If no such value has been set, the call to **pcre_fullinfo()** returns the error PCRE_ERROR_UNSET.

PCRE_INFO_SIZE

Return the size of the compiled pattern in bytes (for all three libraries). The fourth argument should point to a **size_t** variable. This value does not include the size of the **pcre** structure that is returned by **pcre_compile()**. The value that is passed as the argument to **pcre_malloc()** when **pcre_compile()** is getting memory in which to place the compiled data is the value returned by this option plus the size of the **pcre** structure. Studying a compiled pattern, with or without JIT, does not alter the value returned by this option.

PCRE_INFO_STUDYSIZE

Return the size in bytes (for all three libraries) of the data block pointed to by the *study_data* field in a **pcre_extra** block. If **pcre_extra** is NULL, or there is no study data, zero is returned. The fourth argument should point to a **size_t** variable. The *study_data* field is set by **pcre_study()** to record information that will speed up matching (see the section entitled "Studying a pattern" above). The format of the *study_data* block is private, but its length is made available via this option so that it can be saved and restored (see the **pcreprecompile** documentation for details).

PCRE_INFO_REQUIREDCHARFLAGS

Returns 1 if there is a rightmost literal data unit that must exist in any matched string, other than at its

start. The fourth argument should point to an **int** variable. If there is no such value, 0 is returned. If returning 1, the character value itself can be retrieved using PCRE_INFO_REQUIREDCHAR.

For anchored patterns, a last literal value is recorded only if it follows something of variable length. For example, for the pattern /^a\d+z\d+/ the returned value 1 (with "z" returned from PCRE_INFO_REQUIREDCHAR), but for /^a\dz\d/ the returned value is 0.

PCRE_INFO_REQUIREDCHAR

Return the value of the rightmost literal data unit that must exist in any matched string, other than at its start, if such a value has been recorded. The fourth argument should point to a **uint32_t** variable. If there is no such value, 0 is returned.

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 was studied, the result of the study should be passed in the *extra* argument. You can call **pcre_exec()** with the same *code* and *extra* arguments as many times as you like, in order to match different subject strings with the same pattern.

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 below 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(
  re,           /* result of pcre_compile() */
  NULL,         /* we didn't study the pattern */
  "some string",  /* the subject string */
  11,           /* the length of the subject string */
  0,            /* start at offset 0 in the subject */
  0,            /* default options */
  ovector,      /* vector of integers for substring information */
  30);          /* number of elements (NOT size in bytes) */
```

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;
void *executable_jit;
unsigned long int match_limit;
unsigned long int match_limit_recursion;
void *callout_data;
const unsigned char *tables;
unsigned char **mark;
```

In the 16-bit version of this structure, the *mark* field has type "PCRE_UCHAR16 **".

In the 32-bit version of this structure, the *mark* field has type "PCRE_UCHAR32 **".

The *flags* field is used to specify which of the other fields are set. The flag bits are:

```
PCRE_EXTRA_CALLOUT_DATA
PCRE_EXTRA_EXECUTABLE_JIT
PCRE_EXTRA_MARK
PCRE_EXTRA_MATCH_LIMIT
PCRE_EXTRA_MATCH_LIMIT_RECURSION
PCRE_EXTRA_STUDY_DATA
PCRE_EXTRA_TABLES
```

Other flag bits should be set to zero. The *study_data* field and sometimes the *executable_jit* field are set in the **pcre_extra** block that is returned by **pcre_study()**, together with the appropriate flag bits. You should not set these yourself, but you may add to the block by setting 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 a pattern that uses nested unlimited repeats.

Internally, **pcre_exec()** 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.

When **pcre_exec()** is called with a pattern that was successfully studied with a JIT option, the way that the matching is executed is entirely different. However, there is still the possibility of runaway matching that goes on for a very long time, and so the *match_limit* value is also used in this case (but in a different way) to limit how long the matching can continue.

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 supplying **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.

A value for the match limit may also be supplied by an item at the start of a pattern of the form

```
(*LIMIT_MATCH=d)
```

where d is a decimal number. However, such a setting is ignored unless d is less than the limit set by the caller of **pcre_exec()** or, if no such limit is set, less than the default.

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 machine 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. This limit is not relevant, and is ignored, when matching is done using JIT compiled code.

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 supplying **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.

A value for the recursion limit may also be supplied by an item at the start of a pattern of the form

```
(*LIMIT_RECURSION=d)
```

where d is a decimal number. However, such a setting is ignored unless d is less than the limit set by the caller of **pcre_exec()** or, if no such limit is set, less than the default.

The *callout_data* field is used in conjunction with the "callout" feature, and is described in the **pcrecallout** documentation.

The *tables* field is provided for use with patterns that have been pre-compiled using custom character tables, saved to disc or elsewhere, and then reloaded, because the tables that were used to compile a pattern are not saved with it. See the **pcreprecompile** documentation for a discussion of saving compiled patterns for later use. If NULL is passed using this mechanism, it forces PCRE’s internal tables to be used.

Warning: The tables that **pcre_exec()** uses must be the same as those that were used when the pattern was compiled. If this is not the case, the behaviour of **pcre_exec()** is undefined. Therefore, when a pattern is compiled and matched in the same process, this field should never be set. In this (the most common) case, the correct table pointer is automatically passed with the compiled pattern from **pcre_compile()** to **pcre_exec()**.

If PCRE_EXTRA_MARK is set in the *flags* field, the *mark* field must be set to point to a suitable variable. If the pattern contains any backtracking control verbs such as (*MARK:NAME), and the execution ends up with a name to pass back, a pointer to the name string (zero terminated) is placed in the variable pointed to by the *mark* field. The names are within the compiled pattern; if you wish to retain such a name you must copy it before freeing the memory of a compiled pattern. If there is no name to pass back, the variable pointed to by the *mark* field is set to NULL. For details of the backtracking control verbs, see the section entitled "Backtracking control" in the **pcrepattern** documentation.

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_NOTEMPTY_ATSTART, PCRE_NO_START_OPTIMIZE, PCRE_NO_UTF8_CHECK, PCRE_PARTIAL_HARD, and PCRE_PARTIAL_SOFT.

If the pattern was successfully studied with one of the just-in-time (JIT) compile options, the only supported options for JIT execution are PCRE_NO_UTF8_CHECK, PCRE_NOTBOL, PCRE_NOTEOL, PCRE_NOTEMPTY, PCRE_NOTEMPTY_ATSTART, PCRE_PARTIAL_HARD, and PCRE_PARTIAL_SOFT. If an unsupported option is used, JIT execution is disabled and the normal interpretive code in **pcre_exec()** is run.

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_BSR_ANYCRLF
PCRE_BSR_UNICODE

These options (which are mutually exclusive) control what the \R escape sequence matches. The choice is either to match only CR, LF, or CRLF, or to match any Unicode newline sequence. These options override the choice that was made or defaulted when the pattern was compiled.

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 for an unanchored pattern fails when the current position is at a CRLF sequence, and the pattern contains no explicit matches for CR or LF characters, the match position is advanced by two characters instead of one, in other words, to after the CRLF.

The above rule is a compromise that makes the most common cases work as expected. For example, if the pattern is .+A (and the PCRE_DOTALL option is not set), it does not match the string "\r\nA" because, after failing at the start, it skips both the CR and the LF before retrying. However, the pattern [\r\n]A does match that string, because it contains an explicit CR or LF reference, and so advances only by one character after the first failure.

An explicit match for CR of LF is either a literal appearance of one of those characters, or one of the \r or \n escape sequences. Implicit matches such as [^X] do not count, nor does \s (which includes CR and LF in the characters that it matches).

Notwithstanding the above, anomalous effects may still occur when CRLF is a valid newline sequence and explicit \r or \n escapes appear in the pattern.

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 an 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".

PCRE_NOTEMPTY_ATSTART

This is like PCRE_NOTEMPTY, except that an empty string match that is not at the start of the subject is permitted. If the pattern is anchored, such a match can occur only if the pattern contains \K.

Perl has no direct equivalent of PCRE_NOTEMPTY or PCRE_NOTEMPTY_ATSTART, 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_ATSTART 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** sample program. In the most general case, you have to check to see if the newline convention recognizes CRLF as a newline, and if so, and the current character is CR followed by LF, advance the starting offset by two characters instead of one.

PCRE_NO_START_OPTIMIZE

There are a number of optimizations that **pcre_exec()** uses at the start of a match, in order to speed up the process. For example, if it is known that an unanchored match must start with a specific character, it searches the subject for that character, and fails immediately if it cannot find it, without actually running the main matching function. This means that a special item such as (*COMMIT) at the start of a pattern is not considered until after a suitable starting point for the match has been found. Also, when callouts or (*MARK) items are in use, these "start-up" optimizations can cause them to be skipped if the pattern is never actually used. The start-up optimizations are in effect a pre-scan of the subject that takes place before the pattern is run.

The PCRE_NO_START_OPTIMIZE option disables the start-up optimizations, possibly causing performance to suffer, but ensuring that in cases where the result is "no match", the callouts do occur, and that items such as (*COMMIT) and (*MARK) are considered at every possible starting position in the subject string. If PCRE_NO_START_OPTIMIZE is set at compile time, it cannot be unset at matching time. The use of PCRE_NO_START_OPTIMIZE at matching time (that is, passing it to

pcre_exec()) disables JIT execution; in this situation, matching is always done using interpretively.

Setting PCRE_NO_START_OPTIMIZE can change the outcome of a matching operation. Consider the pattern

(*COMMIT)ABC

When this is compiled, PCRE records the fact that a match must start with the character "A". Suppose the subject string is "DEFABC". The start-up optimization scans along the subject, finds "A" and runs the first match attempt from there. The (*COMMIT) item means that the pattern must match the current starting position, which in this case, it does. However, if the same match is run with PCRE_NO_START_OPTIMIZE set, the initial scan along the subject string does not happen. The first match attempt is run starting from "D" and when this fails, (*COMMIT) prevents any further matches being tried, so the overall result is "no match". If the pattern is studied, more start-up optimizations may be used. For example, a minimum length for the subject may be recorded. Consider the pattern

(*MARK:A)(X|Y)

The minimum length for a match is one character. If the subject is "ABC", there will be attempts to match "ABC", "BC", "C", and then finally an empty string. If the pattern is studied, the final attempt does not take place, because PCRE knows that the subject is too short, and so the (*MARK) is never encountered. In this case, studying the pattern does not affect the overall match result, which is still "no match", but it does affect the auxiliary information that is returned.

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 entire string is checked before any other processing takes place. The value of *startoffset* is also checked to ensure that it points to the start of a UTF-8 character. There is a discussion about the validity of UTF-8 strings in the **pcreunicode** page. If an invalid sequence of bytes is found, **pcre_exec()** returns the error PCRE_ERROR_BADUTF8 or, if PCRE_PARTIAL_HARD is set and the problem is a truncated character at the end of the subject, PCRE_ERROR_SHORTUTF8. In both cases, information about the precise nature of the error may also be returned (see the descriptions of these errors in the section entitled *Error return values from* **pcre_exec()** below). If *startoffset* contains a value that does not point to the start of a UTF-8 character (or to the end of the subject), 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 character (or the end of the subject). When PCRE_NO_UTF8_CHECK is set, the effect of passing an invalid string as a subject or an invalid value of *startoffset* is undefined. Your program may crash or loop.

PCRE_PARTIAL_HARD
PCRE_PARTIAL_SOFT

These options turn on the partial matching feature. For backwards compatibility, PCRE_PARTIAL is a synonym for PCRE_PARTIAL_SOFT. A partial match occurs if the end of the subject string is reached successfully, but there are not enough subject characters to complete the match. If this happens when PCRE_PARTIAL_SOFT (but not PCRE_PARTIAL_HARD) is set, matching continues by testing any remaining alternatives. Only if no complete match can be found is PCRE_ERROR_PARTIAL returned instead of PCRE_ERROR_NOMATCH. In other words, PCRE_PARTIAL_SOFT says that the caller is prepared to handle a partial match, but only if no complete match can be found.

If PCRE_PARTIAL_HARD is set, it overrides PCRE_PARTIAL_SOFT. In this case, if a partial match is found, **pcre_exec()** immediately returns PCRE_ERROR_PARTIAL, without considering any other alternatives. In other words, when PCRE_PARTIAL_HARD is set, a partial match is considered to be more important that an alternative complete match.

In both cases, the portion of the string that was inspected when the partial match was found is set as the first matching string. There is a more detailed discussion of partial and multi-segment matching, with examples, 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 offset in *startoffset*. The units for *length* and *startoffset* are bytes for the 8-bit library, 16-bit data items for the 16-bit library, and 32-bit data items for the 32-bit library.

If *startoffset* is negative or greater than the length of the subject, **pcre_exec()** returns PCRE_ERROR_BADOFFSET. 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. In UTF-8 or UTF-16 mode, the offset must point to the start of a character, or the end of the subject (in UTF-32 mode, one data unit equals one character, so all offsets are valid). Unlike the pattern string, the subject may contain binary zeroes.

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 "Mississippi" the first call to **pcre_exec()** finds the first occurrence. If **pcre_exec()** is called again with just the remainder of the subject, namely "issippi", 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.

Finding all the matches in a subject is tricky when the pattern can match an empty string. It is possible to emulate Perl's /g behaviour by first trying the match again at the same offset, with the PCRE_NOTEMPTY_ATSTART and PCRE_ANCHORED options, and then if that fails, advancing the starting offset and trying an ordinary match again. There is some code that demonstrates how to do this in the **pcredemo** sample program. In the most general case, you have to check to see if the newline convention recognizes CRLF as a newline, and if so, and the current character is CR followed by LF, advance the starting offset by two characters instead of one.

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 integers 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 number 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 each 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. These values are always data unit offsets, even in UTF mode. They are byte offsets in the 8-bit library, 16-bit data item offsets in the 16-bit library, and 32-bit data item offsets in the 32-bit library. **Note**: they are not character counts.

The first pair of integers, *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. If neither the actual string matched nor any captured substrings are 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* of reasonable size.

There are some cases where zero is returned (indicating vector overflow) when in fact the vector is exactly the right size for the final match. For example, consider the pattern

```
(a)(?:(b)c|bd)
```

If a vector of 6 elements (allowing for only 1 captured substring) is given with subject string "abd", **pcre_exec()** will try to set the second captured string, thereby recording a vector overflow, before failing to match "c" and backing up to try the second alternative. The zero return, however, does correctly indicate that the maximum number of slots (namely 2) have been filled. In similar cases where there is temporary overflow, but the final number of used slots is actually less than the maximum, a non-zero value is returned.

The **pcre_fullinfo()** 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, and the offsets for for the second and third capturing subpatterns (assuming the vector is large enough, of course) are set to -1.

Note: Elements in the first two-thirds of *ovector* that do not correspond to capturing parentheses in the pattern are never changed. That is, if a pattern contains *n* capturing parentheses, no more than *ovector[0]* to *ovector[2n+1]* are set by **pcre_exec()**. The other elements (in the first two-thirds) retain whatever values they previously had.

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.

This error is also given if **pcre_stack_malloc()** fails in **pcre_exec()**. This can happen only when PCRE has been compiled with **--disable-stack-for-recursion**.

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, and the PCRE_NO_UTF8_CHECK option was not set. If the size of the output vector (*ovecsize*) is at least 2, the byte offset to the start of the the invalid UTF-8 character is placed in the first element, and a reason code is placed in the second element. The reason codes are listed in the following section. For backward compatibility, if PCRE_PARTIAL_HARD is set and the problem is a truncated UTF-8 character at the end of the subject (reason codes 1 to 5), PCRE_ERROR_SHORTUTF8 is returned instead of PCRE_ERROR_BADUTF8.

PCRE_ERROR_BADUTF8_OFFSET (-11)

The UTF-8 byte sequence that was passed as a subject was checked and found to be valid (the PCRE_NO_UTF8_CHECK option was not set), but the value of *startoffset* did not point to the beginning of a UTF-8 character or the end of the subject.

PCRE_ERROR_PARTIAL (-12)

The subject string did not match, but it did match partially. See the **pcrepartial** documentation for details of partial matching.

PCRE_ERROR_BADPARTIAL (-13)

This code is no longer in use. It was formerly returned when the PCRE_PARTIAL option was used with a compiled pattern containing items that were not supported for partial matching. From release 8.00 onwards, there are no restrictions on 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_BADNEWLINE (-23)

An invalid combination of PCRE_NEWLINE_*xxx* options was given.

PCRE_ERROR_BADOFFSET (-24)

The value of *startoffset* was negative or greater than the length of the subject, that is, the value in *length*.

PCRE_ERROR_SHORTUTF8 (-25)

This error is returned instead of PCRE_ERROR_BADUTF8 when the subject string ends with a truncated UTF-8 character and the PCRE_PARTIAL_HARD option is set. Information about the failure is returned as for PCRE_ERROR_BADUTF8. It is in fact sufficient to detect this case, but this special error code for PCRE_PARTIAL_HARD precedes the implementation of returned information; it is retained for backwards compatibility.

PCRE_ERROR_RECURSELOOP (-26)

This error is returned when **pcre_exec()** detects a recursion loop within the pattern. Specifically, it means that either the whole pattern or a subpattern has been called recursively for the second time at the same position in the subject string. Some simple patterns that might do this are detected and faulted at compile time, but more complicated cases, in particular mutual recursions between two different subpatterns, cannot be detected until run time.

PCRE_ERROR_JIT_STACKLIMIT (-27)

This error is returned when a pattern that was successfully studied using a JIT compile option is being matched, but the memory available for the just-in-time processing stack is not large enough. See the **pcrejit** documentation for more details.

PCRE_ERROR_BADMODE (-28)

This error is given if a pattern that was compiled by the 8-bit library is passed to a 16-bit or 32-bit library function, or vice versa.

PCRE_ERROR_BADENDIANNESS (-29)

This error is given if a pattern that was compiled and saved is reloaded on a host with different endianness. The utility function **pcre_pattern_to_host_byte_order()** can be used to convert such a pattern so that it runs on the new host.

PCRE_ERROR_JIT_BADOPTION

This error is returned when a pattern that was successfully studied using a JIT compile option is being matched, but the matching mode (partial or complete match) does not correspond to any JIT compilation mode. When the JIT fast path function is used, this error may be also given for invalid options. See the **pcrejit** documentation for more details.

PCRE_ERROR_BADLENGTH (-32)

This error is given if **pcre_exec()** is called with a negative value for the *length* argument.

Error numbers -16 to -20, -22, and 30 are not used by **pcre_exec()**.

Reason codes for invalid UTF-8 strings

This section applies only to the 8-bit library. The corresponding information for the 16-bit and 32-bit libraries is given in the **pcre16** and **pcre32** pages.

When **pcre_exec()** returns either PCRE_ERROR_BADUTF8 or PCRE_ERROR_SHORTUTF8, and the size of the output vector (*ovecsize*) is at least 2, the offset of the start of the invalid UTF-8 character is placed in the first output vector element (*ovector[0]*) and a reason code is placed in the second element (*ovector[1]*). The reason codes are given names in the **pcre.h** header file:

PCRE_UTF8_ERR1
PCRE_UTF8_ERR2
PCRE_UTF8_ERR3
PCRE_UTF8_ERR4
PCRE_UTF8_ERR5

The string ends with a truncated UTF-8 character; the code specifies how many bytes are missing (1 to 5). Although RFC 3629 restricts UTF-8 characters to be no longer than 4 bytes, the encoding scheme (originally defined by RFC 2279) allows for up to 6 bytes, and this is checked first; hence the possibility of 4 or 5 missing bytes.

PCRE_UTF8_ERR6
PCRE_UTF8_ERR7
PCRE_UTF8_ERR8
PCRE_UTF8_ERR9
PCRE_UTF8_ERR10

The two most significant bits of the 2nd, 3rd, 4th, 5th, or 6th byte of the character do not have the binary value 0b10 (that is, either the most significant bit is 0, or the next bit is 1).

PCRE_UTF8_ERR11
PCRE_UTF8_ERR12

A character that is valid by the RFC 2279 rules is either 5 or 6 bytes long; these code points are excluded by RFC 3629.

PCRE_UTF8_ERR13

A 4-byte character has a value greater than 0x10fff; these code points are excluded by RFC 3629.

PCRE_UTF8_ERR14

A 3-byte character has a value in the range 0xd800 to 0xdfff; this range of code points are reserved by RFC 3629 for use with UTF-16, and so are excluded from UTF-8.

PCRE_UTF8_ERR15
PCRE_UTF8_ERR16
PCRE_UTF8_ERR17
PCRE_UTF8_ERR18
PCRE_UTF8_ERR19

A 2-, 3-, 4-, 5-, or 6-byte character is "overlong", that is, it codes for a value that can be represented by fewer bytes, which is invalid. For example, the two bytes 0xc0, 0xae give the value 0x2e, whose correct coding uses just one byte.

PCRE_UTF8_ERR20

The two most significant bits of the first byte of a character have the binary value 0b10 (that is, the most significant bit is 1 and the second is 0). Such a byte can only validly occur as the second or subsequent byte of a multi-byte character.

PCRE_UTF8_ERR21

The first byte of a character has the value 0xfe or 0xff. These values can never occur in a valid UTF-8 string.

PCRE_UTF8_ERR22

This error code was formerly used when the presence of a so-called "non-character" caused an error. Unicode corrigendum #9 makes it clear that such characters should not cause a string to be rejected, and so this code is no longer in use and is never returned.

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).

Warning: If the pattern uses the (?| feature to set up multiple subpatterns with the same number, as described in the section on duplicate subpattern numbers in the **pcrepattern** page, you cannot use names to distinguish the different subpatterns, because names are not included in the compiled code. The matching process uses only numbers. For this reason, the use of different names for subpatterns of the same number causes an error at compile time.

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. (Duplicate names are always allowed for subpatterns with the same number, created by using the (?| feature. Indeed, if such subpatterns are named, they are required to use the same names.)

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, PCRE_ERROR_NOSUBSTRING (-7) is returned; no data 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* above. 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.

OBTAINING AN ESTIMATE OF STACK USAGE

Matching certain patterns using **pcre_exec()** can use a lot of process stack, which in certain environments can be rather limited in size. Some users find it helpful to have an estimate of the amount of stack that is used by **pcre_exec()**, to help them set recursion limits, as described in the **pcrestack** documentation. The estimate that is output by **pcretest** when called with the **-m** and **-C** options is obtained by calling **pcre_exec** with the values NULL, NULL, NULL, -999, and -999 for its first five arguments.

Normally, if its first argument is NULL, **pcre_exec()** immediately returns the negative error code PCRE_ERROR_NULL, but with this special combination of arguments, it returns instead a negative number whose absolute value is the approximate stack frame size in bytes. (A negative number is used so that it is clear that no match has happened.) The value is approximate because in some cases, recursive calls to **pcre_exec()** occur when there are one or two additional variables on the stack.

If PCRE has been compiled to use the heap instead of the stack for recursion, the value returned is the

size of each block that is obtained from the heap.

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, and a list of features that **pcre_dfa_exec()** does not support, 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()**:

```
int rc;
int ovector[10];
int wspace[20];
rc = pcre_dfa_exec(
  re,           /* result of pcre_compile() */
  NULL,         /* we didn't study the pattern */
  "some string",  /* the subject string */
  11,           /* the length of the subject string */
  0,            /* start at offset 0 in the subject */
  0,            /* default options */
  ovector,      /* vector of integers for substring information */
  10,           /* number of elements (NOT size in bytes) */
  wspace,       /* working space vector */
  20);          /* number of elements (NOT size in bytes) */
```

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_NOTEMPTY_ATSTART, PCRE_NO_UTF8_CHECK, PCRE_BSR_ANYCRLF, PCRE_BSR_UNICODE, PCRE_NO_START_OPTIMIZE, PCRE_PARTIAL_HARD, PCRE_PARTIAL_SOFT, PCRE_DFA_SHORTEST, and PCRE_DFA_RESTART. All but the last four of these are exactly the same as for **pcre_exec()**, so their description is not repeated here.

PCRE_PARTIAL_HARD
PCRE_PARTIAL_SOFT

These have the same general effect as they do for **pcre_exec()**, but the details are slightly different. When PCRE_PARTIAL_HARD is set for **pcre_dfa_exec()**, it returns PCRE_ERROR_PARTIAL if the end of the subject is reached and there is still at least one matching possibility that requires additional characters. This happens even if some complete matches have also been found. When PCRE_PARTIAL_SOFT is set, 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 was inspected when the longest partial match was found is set as the first matching string in both cases. There is a more detailed discussion of partial and multi-segment matching, with examples, in the **pcrepartial** documentation.

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()** 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. Unlike **pcre_exec()**, **pcre_dfa_exec()** can use the entire *ovector* for returning matched strings.

NOTE: PCRE's "auto-possessification" optimization usually applies to character repeats at the end of a pattern (as well as internally). For example, the pattern "a\d+" is compiled as if it were "a\d++" because there is no point even considering the possibility of backtracking into the repeated digits. For DFA matching, this means that only one possible match is found. If you really do want multiple matches in such cases, either use an ungreedy repeat ("a\d+?") or set the PCRE_NO_AUTO_POSSESS option when compiling.

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 above. 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* or *match_limit_recursion* fields. This is not supported (these fields are meaningless for DFA matching).

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.

PCRE_ERROR_DFA_BADRESTART (-30)

When **pcre_dfa_exec()** is called with the **PCRE_DFA_RESTART** option, some plausibility checks are made on the contents of the workspace, which should contain data about the previous partial match. If any of these checks fail, this error is given.

SEE ALSO

pcre16(3), **pcre32**(3), **pcrebuild**(3), **pcrecallout**(3), **pcrecpp(3)**(3), **pcrematching**(3), **pcrepartial**(3), **pcreposix**(3), **pcreprecompile**(3), **pcresample**(3), **pcrestack**(3).

AUTHOR

Philip Hazel
University Computing Service
Cambridge CB2 3QH, England.

REVISION

Last updated: 18 December 2015

Copyright (c) 1997-2015 University of Cambridge.