NAME

PCRE2 - Perl-compatible regular expressions (revised API)

#include <pcre2.h>

PCRE2 is a new API for PCRE, starting at release 10.0. This document contains a description of all its native functions. See the **pcre2** document for an overview of all the PCRE2 documentation.

PCRE2 NATIVE API BASIC FUNCTIONS

pcre2_code *pcre2_compile(PCRE2_SPTR *pattern*, **PCRE2_SIZE** *length*,
uint32_t *options*, **int** **errorcode*, **PCRE2_SIZE** **erroroffset,*
pcre2_compile_context **ccontext*);

void pcre2_code_free(pcre2_code **code*);

pcre2_match_data *pcre2_match_data_create(uint32_t *ovecsize*,
pcre2_general_context **gcontext*);

pcre2_match_data *pcre2_match_data_create_from_pattern(
const pcre2_code **code*, **pcre2_general_context** **gcontext*);

int pcre2_match(const pcre2_code **code*, **PCRE2_SPTR** *subject*,
PCRE2_SIZE *length*, **PCRE2_SIZE** *startoffset*,
uint32_t *options*, **pcre2_match_data** **match_data*,
pcre2_match_context **mcontext*);

int pcre2_dfa_match(const pcre2_code **code*, **PCRE2_SPTR** *subject*,
PCRE2_SIZE *length*, **PCRE2_SIZE** *startoffset*,
uint32_t *options*, **pcre2_match_data** **match_data*,
pcre2_match_context **mcontext*,
int **workspace*, **PCRE2_SIZE** *wscount*);

void pcre2_match_data_free(pcre2_match_data **match_data*);

PCRE2 NATIVE API AUXILIARY MATCH FUNCTIONS

PCRE2_SPTR pcre2_get_mark(pcre2_match_data **match_data*);

uint32_t pcre2_get_ovector_count(pcre2_match_data **match_data*);

PCRE2_SIZE *pcre2_get_ovector_pointer(pcre2_match_data **match_data*);

PCRE2_SIZE pcre2_get_startchar(pcre2_match_data **match_data*);

PCRE2 NATIVE API GENERAL CONTEXT FUNCTIONS

pcre2_general_context *pcre2_general_context_create(
void *(*private_malloc**)(PCRE2_SIZE, void *),**
void (*private_free**)(void *, void *), void** **memory_data*);

pcre2_general_context *pcre2_general_context_copy(
pcre2_general_context **gcontext*);

void pcre2_general_context_free(pcre2_general_context **gcontext*);

PCRE2 NATIVE API COMPILE CONTEXT FUNCTIONS

pcre2_compile_context *pcre2_compile_context_create(
  pcre2_general_context ****gcontext*);**

pcre2_compile_context *pcre2_compile_context_copy(
  pcre2_compile_context ****ccontext*);**

void pcre2_compile_context_free(pcre2_compile_context ****ccontext*);**

int pcre2_set_bsr(pcre2_compile_context ****ccontext*,**
  uint32_t *value***);**

int pcre2_set_character_tables(pcre2_compile_context ****ccontext*,**
  const uint8_t ****tables*);**

int pcre2_set_compile_extra_options(pcre2_compile_context ****ccontext*,**
  uint32_t *extra_options***);**

int pcre2_set_max_pattern_length(pcre2_compile_context ****ccontext*,**
  PCRE2_SIZE *value***);**

int pcre2_set_newline(pcre2_compile_context ****ccontext*,**
  uint32_t *value***);**

int pcre2_set_parens_nest_limit(pcre2_compile_context ****ccontext*,**
  uint32_t *value***);**

int pcre2_set_compile_recursion_guard(pcre2_compile_context ****ccontext*,**
  int (**guard_function***)(uint32_t, void *), void ****user_data***);**

PCRE2 NATIVE API MATCH CONTEXT FUNCTIONS

pcre2_match_context *pcre2_match_context_create(
  pcre2_general_context ****gcontext*);**

pcre2_match_context *pcre2_match_context_copy(
  pcre2_match_context ****mcontext*);**

void pcre2_match_context_free(pcre2_match_context ****mcontext*);**

int pcre2_set_callout(pcre2_match_context ****mcontext*,**
  int (**callout_function***)(pcre2_callout_block *, void *),**
  void ****callout_data*);**

int pcre2_set_substitute_callout(pcre2_match_context ****mcontext*,**
  int (**callout_function***)(pcre2_substitute_callout_block *, void *),**
  void ****callout_data*);**

int pcre2_set_offset_limit(pcre2_match_context ****mcontext*,**
  PCRE2_SIZE *value***);**

int pcre2_set_heap_limit(pcre2_match_context ****mcontext*,**
  uint32_t *value***);**

int pcre2_set_match_limit(pcre2_match_context ***_mcontext_,**
uint32_t _value_**);**

int pcre2_set_depth_limit(pcre2_match_context ***_mcontext_,**
uint32_t _value_**);**

PCRE2 NATIVE API STRING EXTRACTION FUNCTIONS

int pcre2_substring_copy_byname(pcre2_match_data ***_match_data_,**
PCRE2_SPTR _name_**, PCRE2_UCHAR ***_buffer_**, PCRE2_SIZE ***_bufflen_**);**

int pcre2_substring_copy_bynumber(pcre2_match_data ***_match_data_,**
uint32_t _number_**, PCRE2_UCHAR ***_buffer_**,**
PCRE2_SIZE ***_bufflen_);**

void pcre2_substring_free(PCRE2_UCHAR ***_buffer_);**

int pcre2_substring_get_byname(pcre2_match_data ***_match_data_,**
PCRE2_SPTR _name_**, PCRE2_UCHAR ****_bufferptr_**, PCRE2_SIZE ***_bufflen_**);**

int pcre2_substring_get_bynumber(pcre2_match_data ***_match_data_,**
uint32_t _number_**, PCRE2_UCHAR ****_bufferptr_**,**
PCRE2_SIZE ***_bufflen_);**

int pcre2_substring_length_byname(pcre2_match_data ***_match_data_,**
PCRE2_SPTR _name_**, PCRE2_SIZE ***_length_**);**

int pcre2_substring_length_bynumber(pcre2_match_data ***_match_data_,**
uint32_t _number_**, PCRE2_SIZE ***_length_**);**

int pcre2_substring_nametable_scan(const pcre2_code ***_code_,**
PCRE2_SPTR _name_**, PCRE2_SPTR ***_first_**, PCRE2_SPTR ***_last_**);**

int pcre2_substring_number_from_name(const pcre2_code ***_code_,**
PCRE2_SPTR _name_**);**

void pcre2_substring_list_free(PCRE2_SPTR ***_list_);**

int pcre2_substring_list_get(pcre2_match_data ***_match_data_,**
PCRE2_UCHAR *****_listptr_, PCRE2_SIZE ****_lengthsptr_**);**

PCRE2 NATIVE API STRING SUBSTITUTION FUNCTION

int pcre2_substitute(const pcre2_code ***_code_, PCRE2_SPTR** _subject_**,**
PCRE2_SIZE _length_**, PCRE2_SIZE** _startoffset_**,**
uint32_t _options_**, pcre2_match_data ***_match_data_**,**
pcre2_match_context ***_mcontext_, PCRE2_SPTR** _replacementz_**,**
PCRE2_SIZE _rlength_**, PCRE2_UCHAR ***_outputbuffer_**,**
PCRE2_SIZE ***_outlengthptr_);**

PCRE2 NATIVE API JIT FUNCTIONS

int pcre2_jit_compile(pcre2_code ***_code_, uint32_t** _options_**);**

int pcre2_jit_match(const pcre2_code ***_code_, PCRE2_SPTR** _subject_**,**
PCRE2_SIZE _length_**, PCRE2_SIZE** _startoffset_**,**

uint32_t *options*, **pcre2_match_data** *****match_data*,
pcre2_match_context *****mcontext*);

void pcre2_jit_free_unused_memory(pcre2_general_context *****gcontext*);

pcre2_jit_stack *pcre2_jit_stack_create(PCRE2_SIZE *startsize*,
PCRE2_SIZE *maxsize*, **pcre2_general_context** *****gcontext*);

void pcre2_jit_stack_assign(pcre2_match_context *****mcontext*,
pcre2_jit_callback *callback_function*, **void** *****callback_data*);

void pcre2_jit_stack_free(pcre2_jit_stack *****jit_stack*);

PCRE2 NATIVE API SERIALIZATION FUNCTIONS

int32_t pcre2_serialize_decode(pcre2_code ******codes*,
int32_t *number_of_codes*, **const uint8_t** *****bytes*,
pcre2_general_context *****gcontext*);

int32_t pcre2_serialize_encode(const pcre2_code ******codes*,
int32_t *number_of_codes*, **uint8_t** ******serialized_bytes*,
PCRE2_SIZE *****serialized_size*, **pcre2_general_context** *****gcontext*);

void pcre2_serialize_free(uint8_t *****bytes*);

int32_t pcre2_serialize_get_number_of_codes(const uint8_t *****bytes*);

PCRE2 NATIVE API AUXILIARY FUNCTIONS

pcre2_code *pcre2_code_copy(const pcre2_code *****code*);

pcre2_code *pcre2_code_copy_with_tables(const pcre2_code *****code*);

int pcre2_get_error_message(int *errorcode*, **PCRE2_UCHAR** *****buffer*,
PCRE2_SIZE *bufflen*);

const uint8_t *pcre2_maketables(pcre2_general_context *****gcontext*);

void pcre2_maketables_free(pcre2_general_context *****gcontext*,
const uint8_t *****tables*);

int pcre2_pattern_info(const pcre2_code *****code*, **uint32_t** *what*,
void *****where*);

int pcre2_callout_enumerate(const pcre2_code *****code*,
int (*callback*)(pcre2_callout_enumerate_block *, void *),
void *****user_data*);

int pcre2_config(uint32_t *what*, **void** *****where*);

PCRE2 NATIVE API OBSOLETE FUNCTIONS

int pcre2_set_recursion_limit(pcre2_match_context *****mcontext*,
uint32_t *value*);

int pcre2_set_recursion_memory_management(

pcre2_match_context ****mcontext*,**
void *(**private_malloc***)(PCRE2_SIZE, void *),**
void (**private_free***)(void *, void *), void ****memory_data***);**

These functions became obsolete at release 10.30 and are retained only for backward compatibility. They should not be used in new code. The first is replaced by **pcre2_set_depth_limit()**; the second is no longer needed and has no effect (it always returns zero).

PCRE2 EXPERIMENTAL PATTERN CONVERSION FUNCTIONS

pcre2_convert_context *pcre2_convert_context_create(
pcre2_general_context ****gcontext*);**

pcre2_convert_context *pcre2_convert_context_copy(
pcre2_convert_context ****cvcontext*);**

void pcre2_convert_context_free(pcre2_convert_context ****cvcontext*);**

int pcre2_set_glob_escape(pcre2_convert_context ****cvcontext*,**
uint32_t *escape_char***);**

int pcre2_set_glob_separator(pcre2_convert_context ****cvcontext*,**
uint32_t *separator_char***);**

int pcre2_pattern_convert(PCRE2_SPTR *pattern***, PCRE2_SIZE** *length***,**
uint32_t *options***, PCRE2_UCHAR *****buffer***,**
PCRE2_SIZE ****blength*, pcre2_convert_context ****cvcontext***);**

void pcre2_converted_pattern_free(PCRE2_UCHAR ****converted_pattern*);**

These functions provide a way of converting non-PCRE2 patterns into patterns that can be processed by **pcre2_compile()**. This facility is experimental and may be changed in future releases. At present, "globs" and POSIX basic and extended patterns can be converted. Details are given in the **pcre2convert** documentation.

PCRE2 8-BIT, 16-BIT, AND 32-BIT LIBRARIES

There are three PCRE2 libraries, supporting 8-bit, 16-bit, and 32-bit code units, respectively. However, there is just one header file, **pcre2.h**. This contains the function prototypes and other definitions for all three libraries. One, two, or all three can be installed simultaneously. On Unix-like systems the libraries are called **libpcre2-8**, **libpcre2-16**, and **libpcre2-32**, and they can also co-exist with the original PCRE libraries.

Character strings are passed to and from a PCRE2 library as a sequence of unsigned integers in code units of the appropriate width. Every PCRE2 function comes in three different forms, one for each library, for example:

pcre2_compile_8()
pcre2_compile_16()
pcre2_compile_32()

There are also three different sets of data types:

PCRE2_UCHAR8, PCRE2_UCHAR16, PCRE2_UCHAR32
PCRE2_SPTR8, PCRE2_SPTR16, PCRE2_SPTR32

The UCHAR types define unsigned code units of the appropriate widths. For example, PCRE2_UCHAR16 is usually defined as ‘uint16_t’. The SPTR types are constant pointers to the equivalent UCHAR types, that is, they are pointers to vectors of unsigned code units.

Many applications use only one code unit width. For their convenience, macros are defined whose names are the generic forms such as **pcre2_compile()** and PCRE2_SPTR. These macros use the value of the macro PCRE2_CODE_UNIT_WIDTH to generate the appropriate width-specific function and macro names. PCRE2_CODE_UNIT_WIDTH is not defined by default. An application must define it to be 8, 16, or 32 before including **pcre2.h** in order to make use of the generic names.

Applications that use more than one code unit width can be linked with more than one PCRE2 library, but must define PCRE2_CODE_UNIT_WIDTH to be 0 before including **pcre2.h**, and then use the real function names. Any code that is to be included in an environment where the value of PCRE2_CODE_UNIT_WIDTH is unknown should also use the real function names. (Unfortunately, it is not possible in C code to save and restore the value of a macro.)

If PCRE2_CODE_UNIT_WIDTH is not defined before including **pcre2.h**, a compiler error occurs.

When using multiple libraries in an application, you must take care when processing any particular pattern to use only functions from a single library. For example, if you want to run a match using a pattern that was compiled with **pcre2_compile_16()**, you must do so with **pcre2_match_16()**, not **pcre2_match_8()** or **pcre2_match_32()**.

In the function summaries above, and in the rest of this document and other PCRE2 documents, functions and data types are described using their generic names, without the _8, _16, or _32 suffix.

PCRE2 API OVERVIEW

PCRE2 has its own native API, which is described in this document. There are also some wrapper functions for the 8-bit library that correspond to the POSIX regular expression API, but they do not give access to all the functionality of PCRE2. They are described in the **pcre2posix** documentation. Both these APIs define a set of C function calls.

The native API C data types, function prototypes, option values, and error codes are defined in the header file **pcre2.h**, which also contains definitions of PCRE2_MAJOR and PCRE2_MINOR, the major and minor release numbers for the library. Applications can use these to include support for different releases of PCRE2.

In a Windows environment, if you want to statically link an application program against a non-dll PCRE2 library, you must define PCRE2_STATIC before including **pcre2.h**.

The functions **pcre2_compile()** and **pcre2_match()** 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 *pcre2demo.c* in the PCRE2 source distribution. A listing of this program is given in the **pcre2demo** documentation, and the **pcre2sample** documentation describes how to compile and run it.

The compiling and matching functions recognize various options that are passed as bits in an options argument. There are also some more complicated parameters such as custom memory management functions and resource limits that are passed in "contexts" (which are just memory blocks, described below). Simple applications do not need to make use of contexts.

Just-in-time (JIT) compiler support is an optional feature of PCRE2 that can be built in appropriate hardware environments. It greatly speeds up the matching performance of many patterns. Programs can request that it be used if available by calling **pcre2_jit_compile()** after a pattern has been successfully compiled by **pcre2_compile()**. This does nothing if JIT support is not available.

More complicated programs might need to make use of the specialist functions **pcre2_jit_stack_create()**, **pcre2_jit_stack_free()**, and **pcre2_jit_stack_assign()** in order to control the JIT code’s memory usage.

JIT matching is automatically used by **pcre2_match()** if it is available, unless the PCRE2_NO_JIT option is set. There is also a direct interface for JIT matching, which gives improved performance at the expense of less sanity checking. The JIT-specific functions are discussed in the **pcre2jit** documentation.

A second matching function, **pcre2_dfa_match()**, 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 lookaround 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 **pcre2matching** documentation. There is no JIT support for **pcre2_dfa_match()**.

In addition to the main compiling and matching functions, there are convenience functions for extracting captured substrings from a subject string that has been matched by **pcre2_match()**. They are:

pcre2_substring_copy_byname()
pcre2_substring_copy_bynumber()
pcre2_substring_get_byname()
pcre2_substring_get_bynumber()
pcre2_substring_list_get()
pcre2_substring_length_byname()
pcre2_substring_length_bynumber()
pcre2_substring_nametable_scan()
pcre2_substring_number_from_name()

pcre2_substring_free() and **pcre2_substring_list_free()** are also provided, to free memory used for extracted strings. If either of these functions is called with a NULL argument, the function returns immediately without doing anything.

The function **pcre2_substitute()** can be called to match a pattern and return a copy of the subject string with substitutions for parts that were matched.

Functions whose names begin with **pcre2_serialize_** are used for saving compiled patterns on disc or elsewhere, and reloading them later.

Finally, there are functions for finding out information about a compiled pattern (**pcre2_pattern_info()**) and about the configuration with which PCRE2 was built (**pcre2_config()**).

Functions with names ending with **_free()** are used for freeing memory blocks of various sorts. In all cases, if one of these functions is called with a NULL argument, it does nothing.

STRING LENGTHS AND OFFSETS

The PCRE2 API uses string lengths and offsets into strings of code units in several places. These values are always of type PCRE2_SIZE, which is an unsigned integer type, currently always defined as *size_t*. The largest value that can be stored in such a type (that is ˜(PCRE2_SIZE)0) is reserved as a special indicator for zero-terminated strings and unset offsets. Therefore, the longest string that can be handled is one less than this maximum.

NEWLINES

PCRE2 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 PCRE2 is built, a default can be specified. If it is not, the default is set to LF, which is the Unix standard. However, the newline convention can be changed by an application when calling **pcre2_compile()**, or it can be specified by special text at the start of the pattern itself; this overrides any other settings. See the **pcre2pattern** page for details of the special character sequences.

In the PCRE2 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 **pcre2_match()** 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; this has its own separate convention.

MULTITHREADING

In a multithreaded application it is important to keep thread-specific data separate from data that can be shared between threads. The PCRE2 library code itself is thread-safe: it contains no static or global variables. The API is designed to be fairly simple for non-threaded applications while at the same time ensuring that multithreaded applications can use it.

There are several different blocks of data that are used to pass information between the application and the PCRE2 libraries.

The compiled pattern

A pointer to the compiled form of a pattern is returned to the user when **pcre2_compile()** is successful. The data in the compiled pattern is fixed, and does not change when the pattern is matched. Therefore, it is thread-safe, that is, the same compiled pattern can be used by more than one thread simultaneously. For example, an application can compile all its patterns at the start, before forking off multiple threads that use them. However, if the just-in-time (JIT) optimization feature is being used, it needs separate memory stack areas for each thread. See the **pcre2jit** documentation for more details.

In a more complicated situation, where patterns are compiled only when they are first needed, but are still shared between threads, pointers to compiled patterns must be protected from simultaneous writing by multiple threads. This is somewhat tricky to do correctly. If you know that writing to a pointer is atomic in your environment, you can use logic like this:

```
Get a read-only (shared) lock (mutex) for pointer
if (pointer == NULL)
  {
  Get a write (unique) lock for pointer
  if (pointer == NULL) pointer = pcre2_compile(...
  }
Release the lock
Use pointer in pcre2_match()
```

Of course, testing for compilation errors should also be included in the code.

The reason for checking the pointer a second time is as follows: Several threads may have acquired the shared lock and tested the pointer for being NULL, but only one of them will be given the write lock, with the rest kept waiting. The winning thread will compile the pattern and store the result. After this thread releases the write lock, another thread will get it, and if it does not retest pointer for being NULL, will recompile the pattern and overwrite the pointer, creating a memory leak and possibly causing other issues.

In an environment where writing to a pointer may not be atomic, the above logic is not sufficient. The thread that is doing the compiling may be descheduled after writing only part of the pointer, which could cause other threads to use an invalid value. Instead of checking the pointer itself, a separate "pointer is valid" flag (that can be updated atomically) must be used:

```
Get a read-only (shared) lock (mutex) for pointer
if (!pointer_is_valid)
  {
  Get a write (unique) lock for pointer
```

```
    if (!pointer_is_valid)
      {
      pointer = pcre2_compile(...
      pointer_is_valid = TRUE
      }
    }
  Release the lock
  Use pointer in pcre2_match()
```

If JIT is being used, but the JIT compilation is not being done immediately (perhaps waiting to see if the pattern is used often enough), similar logic is required. JIT compilation updates a value within the compiled code block, so a thread must gain unique write access to the pointer before calling **pcre2_jit_compile()**. Alternatively, **pcre2_code_copy()** or **pcre2_code_copy_with_tables()** can be used to obtain a private copy of the compiled code before calling the JIT compiler.

Context blocks

The next main section below introduces the idea of "contexts" in which PCRE2 functions are called. A context is nothing more than a collection of parameters that control the way PCRE2 operates. Grouping a number of parameters together in a context is a convenient way of passing them to a PCRE2 function without using lots of arguments. The parameters that are stored in contexts are in some sense "advanced features" of the API. Many straightforward applications will not need to use contexts.

In a multithreaded application, if the parameters in a context are values that are never changed, the same context can be used by all the threads. However, if any thread needs to change any value in a context, it must make its own thread-specific copy.

Match blocks

The matching functions need a block of memory for storing the results of a match. This includes details of what was matched, as well as additional information such as the name of a (*MARK) setting. Each thread must provide its own copy of this memory.

PCRE2 CONTEXTS

Some PCRE2 functions have a lot of parameters, many of which are used only by specialist applications, for example, those that use custom memory management or non-standard character tables. To keep function argument lists at a reasonable size, and at the same time to keep the API extensible, "uncommon" parameters are passed to certain functions in a **context** instead of directly. A context is just a block of memory that holds the parameter values. Applications that do not need to adjust any of the context parameters can pass NULL when a context pointer is required.

There are three different types of context: a general context that is relevant for several PCRE2 operations, a compile-time context, and a match-time context.

The general context

At present, this context just contains pointers to (and data for) external memory management functions that are called from several places in the PCRE2 library. The context is named ‘general’ rather than specifically ‘memory’ because in future other fields may be added. If you do not want to supply your own custom memory management functions, you do not need to bother with a general context. A general context is created by:

pcre2_general_context *pcre2_general_context_create(
  void *(**private_malloc***)(PCRE2_SIZE, void *),**
  void (**private_free***)(void *, void *), void ****memory_data***);**

The two function pointers specify custom memory management functions, whose prototypes are:

void *private_malloc(PCRE2_SIZE, void *);
void private_free(void *, void *);

Whenever code in PCRE2 calls these functions, the final argument is the value of *memory_data*. Either of the first two arguments of the creation function may be NULL, in which case the system memory management functions *malloc()* and *free()* are used. (This is not currently useful, as there are no other fields in a general context, but in future there might be.) The *private_malloc()* function is used (if supplied) to obtain memory for storing the context, and all three values are saved as part of the context.

Whenever PCRE2 creates a data block of any kind, the block contains a pointer to the *free()* function that matches the *malloc()* function that was used. When the time comes to free the block, this function is called.

A general context can be copied by calling:

pcre2_general_context *pcre2_general_context_copy(
pcre2_general_context ****gcontext*);**

The memory used for a general context should be freed by calling:

void pcre2_general_context_free(pcre2_general_context ****gcontext*);**

If this function is passed a NULL argument, it returns immediately without doing anything.

The compile context

A compile context is required if you want to provide an external function for stack checking during compilation or to change the default values of any of the following compile-time parameters:

What \R matches (Unicode newlines or CR, LF, CRLF only)
PCRE2's character tables
The newline character sequence
The compile time nested parentheses limit
The maximum length of the pattern string
The extra options bits (none set by default)

A compile context is also required if you are using custom memory management. If none of these apply, just pass NULL as the context argument of *pcre2_compile()*.

A compile context is created, copied, and freed by the following functions:

pcre2_compile_context *pcre2_compile_context_create(
pcre2_general_context ****gcontext*);**

pcre2_compile_context *pcre2_compile_context_copy(
pcre2_compile_context ****ccontext*);**

void pcre2_compile_context_free(pcre2_compile_context ****ccontext*);**

A compile context is created with default values for its parameters. These can be changed by calling the following functions, which return 0 on success, or PCRE2_ERROR_BADDATA if invalid data is detected.

int pcre2_set_bsr(pcre2_compile_context ****ccontext*,**
uint32_t *value***);**

The value must be PCRE2_BSR_ANYCRLF, to specify that \R matches only CR, LF, or CRLF, or PCRE2_BSR_UNICODE, to specify that \R matches any Unicode line ending sequence. The value is used by the JIT compiler and by the two interpreted matching functions, *pcre2_match()* and *pcre2_dfa_match()*.

int pcre2_set_character_tables(pcre2_compile_context ****ccontext*,**
const uint8_t ****tables*);**

The value must be the result of a call to **pcre2_maketables()**, whose only argument is a general context. This function builds a set of character tables in the current locale.

int pcre2_set_compile_extra_options(pcre2_compile_context ****ccontext*,**
uint32_t *extra_options***);**

As PCRE2 has developed, almost all the 32 option bits that are available in the *options* argument of **pcre2_compile()** have been used up. To avoid running out, the compile context contains a set of extra option bits which are used for some newer, assumed rarer, options. This function sets those bits. It always sets all the bits (either on or off). It does not modify any existing setting. The available options are defined in the section entitled "Extra compile options" below.

int pcre2_set_max_pattern_length(pcre2_compile_context ****ccontext*,**
PCRE2_SIZE *value***);**

This sets a maximum length, in code units, for any pattern string that is compiled with this context. If the pattern is longer, an error is generated. This facility is provided so that applications that accept patterns from external sources can limit their size. The default is the largest number that a PCRE2_SIZE variable can hold, which is effectively unlimited.

int pcre2_set_newline(pcre2_compile_context ****ccontext*,**
uint32_t *value***);**

This specifies which characters or character sequences are to be recognized as newlines. The value must be one of PCRE2_NEWLINE_CR (carriage return only), PCRE2_NEWLINE_LF (linefeed only), PCRE2_NEWLINE_CRLF (the two-character sequence CR followed by LF), PCRE2_NEWLINE_ANYCRLF (any of the above), PCRE2_NEWLINE_ANY (any Unicode newline sequence), or PCRE2_NEWLINE_NUL (the NUL character, that is a binary zero).

A pattern can override the value set in the compile context by starting with a sequence such as (*CRLF). See the **pcre2pattern** page for details.

When a pattern is compiled with the PCRE2_EXTENDED or PCRE2_EXTENDED_MORE option, the newline convention affects the recognition of the end of internal comments starting with #. The value is saved with the compiled pattern for subsequent use by the JIT compiler and by the two interpreted matching functions, *pcre2_match()* and *pcre2_dfa_match()*.

int pcre2_set_parens_nest_limit(pcre2_compile_context ****ccontext*,**
uint32_t *value***);**

This parameter adjusts the limit, set when PCRE2 is built (default 250), on the depth of parenthesis nesting in a pattern. This limit stops rogue patterns using up too much system stack when being compiled. The limit applies to parentheses of all kinds, not just capturing parentheses.

int pcre2_set_compile_recursion_guard(pcre2_compile_context ****ccontext*,**
int (**guard_function***)(uint32_t, void *), void ****user_data***);**

There is at least one application that runs PCRE2 in threads with very limited system stack, where running out of stack is to be avoided at all costs. The parenthesis limit above cannot take account of how much stack is actually available during compilation. For a finer control, you can supply a function that is called whenever **pcre2_compile()** starts to compile a parenthesized part of a pattern. This function can check the actual stack size (or anything else that it wants to, of course).

The first argument to the callout function gives the current depth of nesting, and the second is user data that is set up by the last argument of **pcre2_set_compile_recursion_guard()**. The callout function should return zero if all is well, or non-zero to force an error.

The match context

A match context is required if you want to:

```
  Set up a callout function
  Set an offset limit for matching an unanchored pattern
  Change the limit on the amount of heap used when matching
  Change the backtracking match limit
  Change the backtracking depth limit
  Set custom memory management specifically for the match
```

If none of these apply, just pass NULL as the context argument of **pcre2_match()**, **pcre2_dfa_match()**, or **pcre2_jit_match()**.

A match context is created, copied, and freed by the following functions:

pcre2_match_context *pcre2_match_context_create(
pcre2_general_context ***gcontext);**

pcre2_match_context *pcre2_match_context_copy(
pcre2_match_context ***mcontext);**

void pcre2_match_context_free(pcre2_match_context ***mcontext);**

A match context is created with default values for its parameters. These can be changed by calling the following functions, which return 0 on success, or PCRE2_ERROR_BADDATA if invalid data is detected.

int pcre2_set_callout(pcre2_match_context ***mcontext,**
int (*callout_function**)(pcre2_callout_block *, void *),**
void ***callout_data);**

This sets up a callout function for PCRE2 to call at specified points during a matching operation. Details are given in the **pcre2callout** documentation.

int pcre2_set_substitute_callout(pcre2_match_context ***mcontext,**
int (*callout_function**)(pcre2_substitute_callout_block *, void *),**
void ***callout_data);**

This sets up a callout function for PCRE2 to call after each substitution made by **pcre2_substitute()**. Details are given in the section entitled "Creating a new string with substitutions" below.

int pcre2_set_offset_limit(pcre2_match_context ***mcontext,**
PCRE2_SIZE *value***);**

The *offset_limit* parameter limits how far an unanchored search can advance in the subject string. The default value is PCRE2_UNSET. The **pcre2_match()** and **pcre2_dfa_match()** functions return PCRE2_ERROR_NOMATCH if a match with a starting point before or at the given offset is not found. The **pcre2_substitute()** function makes no more substitutions.

For example, if the pattern /abc/ is matched against "123abc" with an offset limit less than 3, the result is PCRE2_ERROR_NOMATCH. A match can never be found if the *startoffset* argument of **pcre2_match()**, **pcre2_dfa_match()**, or **pcre2_substitute()** is greater than the offset limit set in the match context.

When using this facility, you must set the PCRE2_USE_OFFSET_LIMIT option when calling **pcre2_compile()** so that when JIT is in use, different code can be compiled. If a match is started with a non-default match limit when PCRE2_USE_OFFSET_LIMIT is not set, an error is generated.

The offset limit facility can be used to track progress when searching large subject strings or to limit the extent of global substitutions. See also the PCRE2_FIRSTLINE option, which requires a match to start before or at the first newline that follows the start of matching in the subject. If this is set with an offset limit, a match must occur in the first line and also within the offset limit. In other words, whichever limit comes first is used.

**int pcre2_set_heap_limit(pcre2_match_context ****mcontext*,
  uint32_t *value***);**

The *heap_limit* parameter specifies, in units of kibibytes (1024 bytes), the maximum amount of heap memory that **pcre2_match()** may use to hold backtracking information when running an interpretive match. This limit also applies to **pcre2_dfa_match()**, which may use the heap when processing patterns with a lot of nested pattern recursion or lookarounds or atomic groups. This limit does not apply to matching with the JIT optimization, which has its own memory control arrangements (see the **pcre2jit** documentation for more details). If the limit is reached, the negative error code PCRE2_ERROR_HEAPLIMIT is returned. The default limit can be set when PCRE2 is built; if it is not, the default is set very large and is essentially "unlimited".

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

```
(*LIMIT_HEAP=ddd)
```

where ddd is a decimal number. However, such a setting is ignored unless ddd is less than the limit set by the caller of **pcre2_match()** or, if no such limit is set, less than the default.

The **pcre2_match()** function starts out using a 20KiB vector on the system stack for recording backtracking points. The more nested backtracking points there are (that is, the deeper the search tree), the more memory is needed. Heap memory is used only if the initial vector is too small. If the heap limit is set to a value less than 21 (in particular, zero) no heap memory will be used. In this case, only patterns that do not have a lot of nested backtracking can be successfully processed.

Similarly, for **pcre2_dfa_match()**, a vector on the system stack is used when processing pattern recursions, lookarounds, or atomic groups, and only if this is not big enough is heap memory used. In this case, too, setting a value of zero disables the use of the heap.

**int pcre2_set_match_limit(pcre2_match_context ****mcontext*,
  uint32_t *value***);**

The *match_limit* parameter provides a means of preventing PCRE2 from using up too many computing resources when processing 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.

There is an internal counter in **pcre2_match()** that is incremented each time round its main matching loop. If this value reaches the match limit, **pcre2_match()** returns the negative value PCRE2_ERROR_MATCHLIMIT. This 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. This limit also applies to **pcre2_dfa_match()**, though the counting is done in a different way.

When **pcre2_match()** is called with a pattern that was successfully processed by **pcre2_jit_compile()**, the way in which 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 PCRE2 is built; the default default is 10 million, which handles all but the most extreme cases. A value for the match limit may also be supplied by an item at the start of a pattern of the form

```
  (*LIMIT_MATCH=ddd)
```

where ddd is a decimal number. However, such a setting is ignored unless ddd is less than the limit set by the caller of **pcre2_match()** or **pcre2_dfa_match()** or, if no such limit is set, less than the default.

int pcre2_set_depth_limit(pcre2_match_context ***_mcontext_,**
  uint32_t _value_**);**

This parameter limits the depth of nested backtracking in **pcre2_match()**. Each time a nested backtracking point is passed, a new memory "frame" is used to remember the state of matching at that point. Thus, this parameter indirectly limits the amount of memory that is used in a match. However, because the size of each memory "frame" depends on the number of capturing parentheses, the actual memory limit varies from pattern to pattern. This limit was more useful in versions before 10.30, where function recursion was used for backtracking.

The depth limit is not relevant, and is ignored, when matching is done using JIT compiled code. However, it is supported by **pcre2_dfa_match()**, which uses it to limit the depth of nested internal recursive function calls that implement atomic groups, lookaround assertions, and pattern recursions. This limits, indirectly, the amount of system stack that is used. It was more useful in versions before 10.32, when stack memory was used for local workspace vectors for recursive function calls. From version 10.32, only local variables are allocated on the stack and as each call uses only a few hundred bytes, even a small stack can support quite a lot of recursion.

If the depth of internal recursive function calls is great enough, local workspace vectors are allocated on the heap from version 10.32 onwards, so the depth limit also indirectly limits the amount of heap memory that is used. A recursive pattern such as /(.(?2))((?1)|)/, when matched to a very long string using **pcre2_dfa_match()**, can use a great deal of memory. However, it is probably better to limit heap usage directly by calling **pcre2_set_heap_limit()**.

The default value for the depth limit can be set when PCRE2 is built; if it is not, the default is set to the same value as the default for the match limit. If the limit is exceeded, **pcre2_match()** or **pcre2_dfa_match()** returns PCRE2_ERROR_DEPTHLIMIT. A value for the depth limit may also be supplied by an item at the start of a pattern of the form

```
  (*LIMIT_DEPTH=ddd)
```

where ddd is a decimal number. However, such a setting is ignored unless ddd is less than the limit set by the caller of **pcre2_match()** or **pcre2_dfa_match()** or, if no such limit is set, less than the default.

CHECKING BUILD-TIME OPTIONS

int pcre2_config(uint32_t _what_**, void ***_where_**);**

The function **pcre2_config()** makes it possible for a PCRE2 client to find the value of certain configuration parameters and to discover which optional features have been compiled into the PCRE2 library. The **pcre2build** documentation has more details about these features.

The first argument for **pcre2_config()** specifies which information is required. The second argument is a pointer to memory into which the information is placed. If NULL is passed, the function returns the amount of memory that is needed for the requested information. For calls that return numerical values, the value is in bytes; when requesting these values, _where_ should point to appropriately aligned memory. For calls that return strings, the required length is given in code units, not counting the terminating zero.

When requesting information, the returned value from **pcre2_config()** is non-negative on success, or the negative error code PCRE2_ERROR_BADOPTION if the value in the first argument is not recognized. The following information is available:

  PCRE2_CONFIG_BSR

The output is a uint32_t integer whose value indicates what character sequences the \R escape sequence matches by default. A value of PCRE2_BSR_UNICODE means that \R matches any Unicode line ending sequence; a value of PCRE2_BSR_ANYCRLF means that \R matches only CR, LF, or CRLF. The default can be overridden when a pattern is compiled.

  PCRE2_CONFIG_COMPILED_WIDTHS

The output is a uint32_t integer whose lower bits indicate which code unit widths were selected when PCRE2 was built. The 1-bit indicates 8-bit support, and the 2-bit and 4-bit indicate 16-bit and 32-bit support, respectively.

  PCRE2_CONFIG_DEPTHLIMIT

The output is a uint32_t integer that gives the default limit for the depth of nested backtracking in **pcre2_match()** or the depth of nested recursions, lookarounds, and atomic groups in **pcre2_dfa_match()**. Further details are given with **pcre2_set_depth_limit()** above.

  PCRE2_CONFIG_HEAPLIMIT

The output is a uint32_t integer that gives, in kibibytes, the default limit for the amount of heap memory used by **pcre2_match()** or **pcre2_dfa_match()**. Further details are given with **pcre2_set_heap_limit()** above.

  PCRE2_CONFIG_JIT

The output is a uint32_t integer that is set to one if support for just-in-time compiling is available; otherwise it is set to zero.

  PCRE2_CONFIG_JITTARGET

The *where* argument should point to a buffer that is at least 48 code units long. (The exact length required can be found by calling **pcre2_config()** with **where** set to NULL.) The buffer is filled with a string that 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, PCRE2_ERROR_BADOPTION is returned, otherwise the number of code units used is returned. This is the length of the string, plus one unit for the terminating zero.

  PCRE2_CONFIG_LINKSIZE

The output is a uint32_t integer that contains the number of bytes used for internal linkage in compiled regular expressions. When PCRE2 is configured, the value can be set to 2, 3, or 4, with the default being 2. This is the value that is returned by **pcre2_config()**. However, when the 16-bit library is compiled, a value of 3 is rounded up to 4, and when the 32-bit library is compiled, internal linkages always use 4 bytes, so the configured value is not relevant.

The default value of 2 for the 8-bit and 16-bit libraries is sufficient for all but the most massive patterns, since it allows the size of the compiled pattern to be up to 65535 code units. Larger values allow larger regular expressions to be compiled by those two libraries, but at the expense of slower matching.

  PCRE2_CONFIG_MATCHLIMIT

The output is a uint32_t integer that gives the default match limit for **pcre2_match()**. Further details are given with **pcre2_set_match_limit()** above.

```
  PCRE2_CONFIG_NEWLINE
```

The output is a uint32_t integer whose value specifies the default character sequence that is recognized as meaning "newline". The values are:

PCRE2_NEWLINE_CR	Carriage return (CR)
PCRE2_NEWLINE_LF	Linefeed (LF)
PCRE2_NEWLINE_CRLF	Carriage return, linefeed (CRLF)
PCRE2_NEWLINE_ANY	Any Unicode line ending
PCRE2_NEWLINE_ANYCRLF	Any of CR, LF, or CRLF
PCRE2_NEWLINE_NUL	The NUL character (binary zero)

The default should normally correspond to the standard sequence for your operating system.

```
  PCRE2_CONFIG_NEVER_BACKSLASH_C
```

The output is a uint32_t integer that is set to one if the use of \C was permanently disabled when PCRE2 was built; otherwise it is set to zero.

```
  PCRE2_CONFIG_PARENSLIMIT
```

The output is a uint32_t 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 PCRE2 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, see **pcre2_set_compile_recursion_guard()**.

```
  PCRE2_CONFIG_STACKRECURSE
```

This parameter is obsolete and should not be used in new code. The output is a uint32_t integer that is always set to zero.

```
  PCRE2_CONFIG_TABLES_LENGTH
```

The output is a uint32_t integer that gives the length of PCRE2's character processing tables in bytes. For details of these tables see the section on locale support below.

```
  PCRE2_CONFIG_UNICODE_VERSION
```

The *where* argument should point to a buffer that is at least 24 code units long. (The exact length required can be found by calling **pcre2_config()** with **where** set to NULL.) If PCRE2 has been compiled without Unicode support, the buffer is filled with the text "Unicode not supported". Otherwise, the Unicode version string (for example, "8.0.0") is inserted. The number of code units used is returned. This is the length of the string plus one unit for the terminating zero.

```
  PCRE2_CONFIG_UNICODE
```

The output is a uint32_t integer that is set to one if Unicode support is available; otherwise it is set to zero. Unicode support implies UTF support.

```
  PCRE2_CONFIG_VERSION
```

The *where* argument should point to a buffer that is at least 24 code units long. (The exact length required can be found by calling **pcre2_config()** with **where** set to NULL.) The buffer is filled with the PCRE2 version string, zero-terminated. The number of code units used is returned. This is the length of the string plus one unit for the terminating zero.

COMPILING A PATTERN

pcre2_code *pcre2_compile(PCRE2_SPTR *pattern*, **PCRE2_SIZE** *length*,
uint32_t *options*, **int** **errorcode*, **PCRE2_SIZE** **erroroffset,*
pcre2_compile_context **ccontext***);**

void pcre2_code_free(pcre2_code **code***);**

pcre2_code *pcre2_code_copy(const pcre2_code **code***);**

pcre2_code *pcre2_code_copy_with_tables(const pcre2_code **code***);**

The **pcre2_compile()** function compiles a pattern into an internal form. The pattern is defined by a pointer to a string of code units and a length (in code units). If the pattern is zero-terminated, the length can be specified as PCRE2_ZERO_TERMINATED. The function returns a pointer to a block of memory that contains the compiled pattern and related data, or NULL if an error occurred.

If the compile context argument *ccontext* is NULL, memory for the compiled pattern is obtained by calling **malloc()**. Otherwise, it is obtained from the same memory function that was used for the compile context. The caller must free the memory by calling **pcre2_code_free()** when it is no longer needed. If **pcre2_code_free()** is called with a NULL argument, it returns immediately, without doing anything.

The function **pcre2_code_copy()** makes a copy of the compiled code in new memory, using the same memory allocator as was used for the original. However, if the code has been processed by the JIT compiler (see below), the JIT information cannot be copied (because it is position-dependent). The new copy can initially be used only for non-JIT matching, though it can be passed to **pcre2_jit_compile()** if required. If **pcre2_code_copy()** is called with a NULL argument, it returns NULL.

The **pcre2_code_copy()** function provides a way for individual threads in a multithreaded application to acquire a private copy of shared compiled code. However, it does not make a copy of the character tables used by the compiled pattern; the new pattern code points to the same tables as the original code. (See "Locale Support" below for details of these character tables.) In many applications the same tables are used throughout, so this behaviour is appropriate. Nevertheless, there are occasions when a copy of a compiled pattern and the relevant tables are needed. The **pcre2_code_copy_with_tables()** provides this facility. Copies of both the code and the tables are made, with the new code pointing to the new tables. The memory for the new tables is automatically freed when **pcre2_code_free()** is called for the new copy of the compiled code. If **pcre2_code_copy_with_tables()** is called with a NULL argument, it returns NULL.

NOTE: When one of the matching functions is called, pointers to the compiled pattern and the subject string are set in the match data block so that they can be referenced by the substring extraction functions after a successful match. After running a match, you must not free a compiled pattern or a subject string until after all operations on the match data block have taken place, unless, in the case of the subject string, you have used the PCRE2_COPY_MATCHED_SUBJECT option, which is described in the section entitled "Option bits for **pcre2_match()**" below.

The *options* argument for **pcre2_compile()** contains various bit settings that affect the compilation. It should be zero if none of them 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 **pcre2pattern** 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. The PCRE2_ANCHORED, PCRE2_ENDANCHORED, and PCRE2_NO_UTF_CHECK options can be set at the time of matching as well as at compile time.

Some additional options and less frequently required compile-time parameters (for example, the newline setting) can be provided in a compile context (as described above).

If *errorcode* or *erroroffset* is NULL, **pcre2_compile()** returns NULL immediately. Otherwise, the variables to which these point are set to an error code and an offset (number of code units) within the pattern, respectively, when **pcre2_compile()** returns NULL because a compilation error has occurred. The values are not defined when compilation is successful and **pcre2_compile()** returns a non-NULL value.

There are nearly 100 positive error codes that **pcre2_compile()** may return if it finds an error in the pattern. There are also some negative error codes that are used for invalid UTF strings when validity checking is in force. These are the same as given by **pcre2_match()** and **pcre2_dfa_match()**, and are described in the **pcre2unicode** documentation. There is no separate documentation for the positive error codes, because the textual error messages that are obtained by calling the **pcre2_get_error_message()** function (see "Obtaining a textual error message" below) should be self-explanatory. Macro names starting with PCRE2_ERROR_ are defined for both positive and negative error codes in **pcre2.h**.

The value returned in *erroroffset* is an indication of where in the pattern the error occurred. It is not necessarily the furthest point in the pattern that was read. For example, after the error "lookbehind assertion is not fixed length", the error offset points to the start of the failing assertion. For an invalid UTF-8 or UTF-16 string, the offset is that of the first code 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 code units, not characters, even in a UTF mode. It may sometimes point into the middle of a UTF-8 or UTF-16 character.

This code fragment shows a typical straightforward call to **pcre2_compile()**:

```
pcre2_code *re;
PCRE2_SIZE erroffset;
int errorcode;
re = pcre2_compile(
  "^A.*Z",                /* the pattern */
  PCRE2_ZERO_TERMINATED,  /* the pattern is zero-terminated */
  0,                      /* default options */
  &errorcode,             /* for error code */
  &erroffset,             /* for error offset */
  NULL);                  /* no compile context */
```

Main compile options

The following names for option bits are defined in the **pcre2.h** header file:

```
PCRE2_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.

```
PCRE2_ALLOW_EMPTY_CLASS
```

By default, for compatibility with Perl, a closing square bracket that immediately follows an opening one is treated as a data character for the class. When PCRE2_ALLOW_EMPTY_CLASS is set, it terminates the class, which therefore contains no characters and so can never match.

```
PCRE2_ALT_BSUX
```

This option request alternative handling of three escape sequences, which makes PCRE2's behaviour more like ECMAscript (aka JavaScript). When it is set:

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

upper case subsequent characters).

(2) \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).

(3) \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).

ECMAscript 6 added additional functionality to \u. This can be accessed using the PCRE2_EXTRA_ALT_BSUX extra option (see "Extra compile options" below). Note that this alternative escape handling applies only to patterns. Neither of these options affects the processing of replacement strings passed to **pcre2_substitute()**.

```
  PCRE2_ALT_CIRCUMFLEX
```

In multiline mode (when PCRE2_MULTILINE is set), the circumflex metacharacter matches at the start of the subject (unless PCRE2_NOTBOL is set), and also after any internal newline. However, it does not match after a newline at the end of the subject, for compatibility with Perl. If you want a multiline circumflex also to match after a terminating newline, you must set PCRE2_ALT_CIRCUMFLEX.

```
  PCRE2_ALT_VERBNAMES
```

By default, for compatibility with Perl, the name in any verb sequence such as (*MARK:NAME) is any sequence of characters that does not include a closing parenthesis. The name is not processed in any way, and it is not possible to include a closing parenthesis in the name. However, if the PCRE2_ALT_VERBNAMES option is set, normal backslash processing is applied to verb names and only an unescaped closing parenthesis terminates the name. A closing parenthesis can be included in a name either as \) or between \Q and \E. If the PCRE2_EXTENDED or PCRE2_EXTENDED_MORE option is set with PCRE2_ALT_VERBNAMES, unescaped whitespace in verb names is skipped and #-comments are recognized, exactly as in the rest of the pattern.

```
  PCRE2_AUTO_CALLOUT
```

If this bit is set, **pcre2_compile()** automatically inserts callout items, all with number 255, before each pattern item, except immediately before or after an explicit callout in the pattern. For discussion of the callout facility, see the **pcre2callout** documentation.

```
  PCRE2_CASELESS
```

If this bit is set, letters in the pattern match both upper and lower case letters in the subject. It is equivalent to Perl's /i option, and it can be changed within a pattern by a (?i) option setting. If either PCRE2_UTF or PCRE2_UCP is set, Unicode properties are used for all characters with more than one other case, and for all characters whose code points are greater than U+007F. Note that there are two ASCII characters, K and S, that, in addition to their lower case ASCII equivalents, are case-equivalent with U+212A (Kelvin sign) and U+017F (long S) respectively. For lower valued characters with only one other case, a lookup table is used for speed. When neither PCRE2_UTF nor PCRE2_UCP is set, a lookup table is used for all code points less than 256, and higher code points (available only in 16-bit or 32-bit mode) are treated as not having another case.

```
  PCRE2_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 PCRE2_DOLLAR_ENDONLY option is ignored if PCRE2_MULTILINE is set. There is no equivalent to this option in Perl, and no way to set it within a pattern.

```
  PCRE2_DOTALL
```

If this bit is set, a dot metacharacter in the pattern matches any character, 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 in the subject 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 [ˆa] always matches newline characters, and the \N escape sequence always matches a non-newline character, independent of the setting of PCRE2_DOTALL.

```
  PCRE2_DUPNAMES
```

If this bit is set, names used to identify capture groups need not be unique. This can be helpful for certain types of pattern when it is known that only one instance of the named group can ever be matched. There are more details of named capture groups below; see also the **pcre2pattern** documentation.

```
  PCRE2_ENDANCHORED
```

If this bit is set, the end of any pattern match must be right at the end of the string being searched (the "subject string"). If the pattern match succeeds by reaching (*ACCEPT), but does not reach the end of the subject, the match fails at the current starting point. For unanchored patterns, a new match is then tried at the next starting point. However, if the match succeeds by reaching the end of the pattern, but not the end of the subject, backtracking occurs and an alternative match may be found. Consider these two patterns:

```
  .(*ACCEPT)|..
  .|..
```

If matched against "abc" with PCRE2_ENDANCHORED set, the first matches "c" whereas the second matches "bc". The effect of PCRE2_ENDANCHORED can also be achieved by appropriate constructs in the pattern itself, which is the only way to do it in Perl.

For DFA matching with **pcre2_dfa_match()**, PCRE2_ENDANCHORED applies only to the first (that is, the longest) matched string. Other parallel matches, which are necessarily substrings of the first one, must obviously end before the end of the subject.

```
  PCRE2_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 groups, nor within numerical quantifiers such as {1,3}. Ignorable white space is permitted between an item and a following quantifier and between a quantifier and a following + that indicates possessiveness. PCRE2_EXTENDED is equivalent to Perl's /x option, and it can be changed within a pattern by a (?x) option setting.

When PCRE2 is compiled without Unicode support, PCRE2_EXTENDED recognizes as white space only those characters with code points less than 256 that are flagged as white space in its low-character table. The table is normally created by **pcre2_maketables()**, which uses the **isspace()** function to identify space characters. In most ASCII environments, the relevant characters are those with code points 0x0009 (tab), 0x000A (linefeed), 0x000B (vertical tab), 0x000C (formfeed), 0x000D (carriage return), and 0x0020 (space).

When PCRE2 is compiled with Unicode support, in addition to these characters, five more Unicode "Pattern White Space" characters are recognized by PCRE2_EXTENDED. These are U+0085 (next line), U+200E (left-to-right mark), U+200F (right-to-left mark), U+2028 (line separator), and U+2029 (paragraph separator). This set of characters is the same as recognized by Perl's /x option. Note that the horizontal and vertical space characters that are matched by the \h and \v escapes in patterns are a much bigger set.

As well as ignoring most white space, PCRE2_EXTENDED also causes characters between an unescaped # outside a character class and the next newline, inclusive, to be ignored, which makes it possible to include comments inside complicated patterns. 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.

Which characters are interpreted as newlines can be specified by a setting in the compile context that is passed to **pcre2_compile()** or by a special sequence at the start of the pattern, as described in the section entitled "Newline conventions" in the **pcre2pattern** documentation. A default is defined when PCRE2 is built.

```
  PCRE2_EXTENDED_MORE
```

This option has the effect of PCRE2_EXTENDED, but, in addition, unescaped space and horizontal tab characters are ignored inside a character class. Note: only these two characters are ignored, not the full set of pattern white space characters that are ignored outside a character class. PCRE2_EXTENDED_MORE is equivalent to Perl's /xx option, and it can be changed within a pattern by a (?xx) option setting.

```
  PCRE2_FIRSTLINE
```

If this option is set, the start of an unanchored pattern match must be before or at the first newline in the subject string following the start of matching, though the matched text may continue over the newline. If *startoffset* is non-zero, the limiting newline is not necessarily the first newline in the subject. For example, if the subject string is "abc\nxyz" (where \n represents a single-character newline) a pattern match for "yz" succeeds with PCRE2_FIRSTLINE if *startoffset* is greater than 3. See also PCRE2_USE_OFFSET_LIMIT, which provides a more general limiting facility. If PCRE2_FIRSTLINE is set with an offset limit, a match must occur in the first line and also within the offset limit. In other words, whichever limit comes first is used.

```
  PCRE2_LITERAL
```

If this option is set, all meta-characters in the pattern are disabled, and it is treated as a literal string. Matching literal strings with a regular expression engine is not the most efficient way of doing it. If you are doing a lot of literal matching and are worried about efficiency, you should consider using other approaches. The only other main options that are allowed with PCRE2_LITERAL are: PCRE2_ANCHORED, PCRE2_ENDANCHORED, PCRE2_AUTO_CALLOUT, PCRE2_CASELESS, PCRE2_FIRSTLINE, PCRE2_MATCH_INVALID_UTF, PCRE2_NO_START_OPTIMIZE, PCRE2_NO_UTF_CHECK, PCRE2_UTF, and PCRE2_USE_OFFSET_LIMIT. The extra options PCRE2_EXTRA_MATCH_LINE and PCRE2_EXTRA_MATCH_WORD are also supported. Any other options cause an error.

```
  PCRE2_MATCH_INVALID_UTF
```

This option forces PCRE2_UTF (see below) and also enables support for matching by **pcre2_match()** in subject strings that contain invalid UTF sequences. This facility is not supported for DFA matching. For details, see the **pcre2unicode** documentation.

```
  PCRE2_MATCH_UNSET_BACKREF
```

If this option is set, a backreference to an unset capture 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. Setting this option makes PCRE2 behave more like ECMAscript (aka JavaScript).

```
  PCRE2_MULTILINE
```

By default, for the purposes of matching "start of line" and "end of line", PCRE2 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 PCRE2_DOLLAR_ENDONLY is set). Note, however, that unless PCRE2_DOTALL is set, the "any character" metacharacter (.) does not match at a newline. This behaviour (for ^, $, and dot) is the same as Perl.

When PCRE2_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. Note that the "start of line" metacharacter does not match after a newline at the end of the subject, for compatibility with Perl. However, you can change this by setting the PCRE2_ALT_CIRCUMFLEX option. If there are no newlines in a subject string, or no occurrences of ^ or $ in a pattern, setting PCRE2_MULTILINE has no effect.

  PCRE2_NEVER_BACKSLASH_C

This option locks out the use of \C in the pattern that is being compiled. This escape can cause unpredictable behaviour in UTF-8 or UTF-16 modes, because it may leave the current matching point in the middle of a multi-code-unit character. This option may be useful in applications that process patterns from external sources. Note that there is also a build-time option that permanently locks out the use of \C.

  PCRE2_NEVER_UCP

This option locks out the use of Unicode properties for handling \B, \b, \D, \d, \S, \s, \W, \w, and some of the POSIX character classes, as described for the PCRE2_UCP option below. In particular, it prevents the creator of the pattern from enabling this facility by starting the pattern with (*UCP). This option may be useful in applications that process patterns from external sources. The option combination PCRE_UCP and PCRE_NEVER_UCP causes an error.

  PCRE2_NEVER_UTF

This option locks out interpretation of the pattern as UTF-8, UTF-16, or UTF-32, depending on which library is in use. In particular, it prevents the creator of the pattern from switching to UTF interpretation by starting the pattern with (*UTF). This option may be useful in applications that process patterns from external sources. The combination of PCRE2_UTF and PCRE2_NEVER_UTF causes an error.

  PCRE2_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). This is the same as Perl's /n option. Note that, when this option is set, references to capture groups (backreferences or recursion/subroutine calls) may only refer to named groups, though the reference can be by name or by number.

  PCRE2_NO_AUTO_POSSESS

If this option is set, it disables "auto-possessification", which 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 callouts 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.

  PCRE2_NO_DOTSTAR_ANCHOR

If this option is set, it disables an optimization that is applied when .* is the first significant item in a top-level branch of a pattern, and all the other branches also start with .* or with \A or \G or ^. The optimization is automatically disabled for .* if it is inside an atomic group or a capture group that is the subject of a backreference, or if the pattern contains (*PRUNE) or (*SKIP). When the optimization is not disabled, such a pattern is automatically anchored if PCRE2_DOTALL is set for all the .* items and PCRE2_MULTILINE is not set for any ^ items. Otherwise, the fact that any match must start either at the start of the subject or following a newline is remembered. Like other optimizations, this can cause callouts to be skipped.

```
  PCRE2_NO_START_OPTIMIZE
```

This is an option whose main effect is at matching time. It does not change what **pcre2_compile()** generates, but it does affect the output of the JIT compiler.

There are a number of optimizations that may occur 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 code unit value, the matching code searches the subject for that value, 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 PCRE2_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.

Setting PCRE2_NO_START_OPTIMIZE may change the outcome of a matching operation. Consider the pattern

```
  (*COMMIT)ABC
```

When this is compiled, PCRE2 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 PCRE2_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".

As another start-up optimization makes use of a minimum length for a matching subject, which is recorded when possible. Consider the pattern

```
  (*MARK:1)B(*MARK:2)(X|Y)
```

The minimum length for a match is two characters. If the subject is "XXBB", the "starting character" optimization skips "XX", then tries to match "BB", which is long enough. In the process, (*MARK:2) is encountered and remembered. When the match attempt fails, the next "B" is found, but there is only one character left, so there are no more attempts, and "no match" is returned with the "last mark seen" set to "2". If NO_START_OPTIMIZE is set, however, matches are tried at every possible starting position, including at the end of the subject, where (*MARK:1) is encountered, but there is no "B", so the "last mark seen" that is returned is "1". In this case, the optimizations do not affect the overall match result, which is still "no match", but they do affect the auxiliary information that is returned.

```
  PCRE2_NO_UTF_CHECK
```

When PCRE2_UTF is set, the validity of the pattern as a UTF string is automatically checked. There are discussions about the validity of UTF-8 strings, UTF-16 strings, and UTF-32 strings in the **pcre2unicode** document. If an invalid UTF sequence is found, **pcre2_compile()** returns a negative error code.

If you know that your pattern is a valid UTF string, and you want to skip this check for performance reasons, you can set the PCRE2_NO_UTF_CHECK option. When it is set, the effect of passing an invalid UTF string as a pattern is undefined. It may cause your program to crash or loop.

Note that this option can also be passed to **pcre2_match()** and **pcre_dfa_match()**, to suppress UTF validity checking of the subject string.

Note also that setting PCRE2_NO_UTF_CHECK at compile time does not disable the error that is given if an escape sequence for an invalid Unicode code point is encountered in the pattern. In particular, the so-called "surrogate" code points (0xd800 to 0xdfff) are invalid. If you want to allow escape sequences such as \x{d800} you can set the PCRE2_EXTRA_ALLOW_SURROGATE_ESCAPES extra option, as described in the section entitled "Extra compile options" below. However, this is possible only in UTF-8 and UTF-32 modes, because these values are not representable in UTF-16.

```
  PCRE2_UCP
```

This option has two effects. Firstly, it change the way PCRE2 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 PCRE2_UCP is set, Unicode properties are used instead to classify characters. More details are given in the section on generic character types in the **pcre2pattern** page. If you set PCRE2_UCP, matching one of the items it affects takes much longer.

The second effect of PCRE2_UCP is to force the use of Unicode properties for upper/lower casing operations on characters with code points greater than 127, even when PCRE2_UTF is not set. This makes it possible, for example, to process strings in the 16-bit UCS-2 code. This option is available only if PCRE2 has been compiled with Unicode support (which is the default).

```
  PCRE2_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.

```
  PCRE2_USE_OFFSET_LIMIT
```

This option must be set for **pcre2_compile()** if **pcre2_set_offset_limit()** is going to be used to set a non-default offset limit in a match context for matches that use this pattern. An error is generated if an offset limit is set without this option. For more details, see the description of **pcre2_set_offset_limit()** in the section that describes match contexts. See also the PCRE2_FIRSTLINE option above.

```
  PCRE2_UTF
```

This option causes PCRE2 to regard both the pattern and the subject strings that are subsequently processed as strings of UTF characters instead of single-code-unit strings. It is available when PCRE2 is built to include Unicode support (which is the default). If Unicode support is not available, the use of this option provokes an error. Details of how PCRE2_UTF changes the behaviour of PCRE2 are given in the **pcre2unicode** page. In particular, note that it changes the way PCRE2_CASELESS handles characters with code points greater than 127.

Extra compile options

The option bits that can be set in a compile context by calling the **pcre2_set_compile_extra_options()** function are as follows:

  PCRE2_EXTRA_ALLOW_LOOKAROUND_BSK

Since release 10.38 PCRE2 has forbidden the use of \K within lookaround assertions, following Perl's lead. This option is provided to re-enable the previous behaviour (act in positive lookarounds, ignore in negative ones) in case anybody is relying on it.

  PCRE2_EXTRA_ALLOW_SURROGATE_ESCAPES

This option applies when compiling a pattern in UTF-8 or UTF-32 mode. It is forbidden in UTF-16 mode, and ignored in non-UTF modes. Unicode "surrogate" code points in the range 0xd800 to 0xdfff are used in pairs in UTF-16 to encode code points with values in the range 0x10000 to 0x10ffff. The surrogates cannot therefore be represented in UTF-16. They can be represented in UTF-8 and UTF-32, but are defined as invalid code points, and cause errors if encountered in a UTF-8 or UTF-32 string that is being checked for validity by PCRE2.

These values also cause errors if encountered in escape sequences such as \x{d912} within a pattern. However, it seems that some applications, when using PCRE2 to check for unwanted characters in UTF-8 strings, explicitly test for the surrogates using escape sequences. The PCRE2_NO_UTF_CHECK option does not disable the error that occurs, because it applies only to the testing of input strings for UTF validity.

If the extra option PCRE2_EXTRA_ALLOW_SURROGATE_ESCAPES is set, surrogate code point values in UTF-8 and UTF-32 patterns no longer provoke errors and are incorporated in the compiled pattern. However, they can only match subject characters if the matching function is called with PCRE2_NO_UTF_CHECK set.

  PCRE2_EXTRA_ALT_BSUX

The original option PCRE2_ALT_BSUX causes PCRE2 to process \U, \u, and \x in the way that ECMAscript (aka JavaScript) does. Additional functionality was defined by ECMAscript 6; setting PCRE2_EXTRA_ALT_BSUX has the effect of PCRE2_ALT_BSUX, but in addition it recognizes \u{hhh..} as a hexadecimal character code, where hhh.. is any number of hexadecimal digits.

  PCRE2_EXTRA_BAD_ESCAPE_IS_LITERAL

This is a dangerous option. Use with care. By default, an unrecognized escape such as \j or a malformed one such as \x{2z} causes a compile-time error when detected by **pcre2_compile()**. Perl is somewhat inconsistent in handling such items: for example, \j is treated as a literal "j", and non-hexadecimal digits in \x{} are just ignored, though warnings are given in both cases if Perl's warning switch is enabled. However, a malformed octal number after \o{ always causes an error in Perl.

If the PCRE2_EXTRA_BAD_ESCAPE_IS_LITERAL extra option is passed to **pcre2_compile()**, all unrecognized or malformed escape sequences are treated as single-character escapes. For example, \j is a literal "j" and \x{2z} is treated as the literal string "x{2z}". Setting this option means that typos in patterns may go undetected and have unexpected results. Also note that a sequence such as [\N{] is interpreted as a malformed attempt at [\N{...}] and so is treated as [N{] whereas [\N] gives an error because an unqualified \N is a valid escape sequence but is not supported in a character class. To reiterate: this is a dangerous option. Use with great care.

  PCRE2_EXTRA_ESCAPED_CR_IS_LF

There are some legacy applications where the escape sequence \r in a pattern is expected to match a newline. If this option is set, \r in a pattern is converted to \n so that it matches a LF (linefeed) instead of a CR (carriage return) character. The option does not affect a literal CR in the pattern, nor does it affect CR specified as an explicit code point such as \x{0D}.

  PCRE2_EXTRA_MATCH_LINE

This option is provided for use by the **-x** option of **pcre2grep**. It causes the pattern only to match complete lines. This is achieved by automatically inserting the code for "ˆ(?:" at the start of the compiled pattern and ")$" at the end. Thus, when PCRE2_MULTILINE is set, the matched line may be in the middle of the subject string. This option can be used with PCRE2_LITERAL.

  PCRE2_EXTRA_MATCH_WORD

This option is provided for use by the **-w** option of **pcre2grep**. It causes the pattern only to match strings that have a word boundary at the start and the end. This is achieved by automatically inserting the code for "\b(?:" at the start of the compiled pattern and ")\b" at the end. The option may be used with PCRE2_LITERAL. However, it is ignored if PCRE2_EXTRA_MATCH_LINE is also set.

JUST-IN-TIME (JIT) COMPILATION

int pcre2_jit_compile(pcre2_code ****code*, **uint32_t *options***);**

int pcre2_jit_match(const pcre2_code ****code*, **PCRE2_SPTR *subject*,
  PCRE2_SIZE *length*, **PCRE2_SIZE** *startoffset*,
  uint32_t *options*, **pcre2_match_data ****match_data*,
  pcre2_match_context ****mcontext*);**

void pcre2_jit_free_unused_memory(pcre2_general_context ****gcontext*);**

pcre2_jit_stack *pcre2_jit_stack_create(PCRE2_SIZE *startsize*,
  PCRE2_SIZE *maxsize*, **pcre2_general_context ****gcontext***);**

**void pcre2_jit_stack_assign(pcre2_match_context ****mcontext*,
  pcre2_jit_callback *callback_function*, **void ****callback_data***);**

void pcre2_jit_stack_free(pcre2_jit_stack ****jit_stack*);**

These functions provide support for JIT compilation, which, if the just-in-time compiler is available, further processes a compiled pattern into machine code that executes much faster than the **pcre2_match()** interpretive matching function. Full details are given in the **pcre2jit** documentation.

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 compilation time. Most (but not all) patterns can be optimized by the JIT compiler.

LOCALE SUPPORT

const uint8_t *pcre2_maketables(pcre2_general_context ****gcontext*);**

**void pcre2_maketables_free(pcre2_general_context ****gcontext*,
  const uint8_t ****tables*);**

PCRE2 handles caseless matching, and determines whether characters are letters, digits, or whatever, by reference to a set of tables, indexed by character code point. However, this applies only to characters whose code points are less than 256. By default, higher-valued code points never match escapes such as \w or \d.

When PCRE2 is built with Unicode support (the default), the Unicode properties of all characters can be tested with \p and \P, or, alternatively, the PCRE2_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. PCRE2_UCP also causes upper/lower casing operations on characters with code points greater than 127 to use Unicode properties. These effects apply even when PCRE2_UTF is not set.

The use of locales with Unicode is discouraged. If you are handling characters with code points greater than 127, you should either use Unicode support, or use locales, but not try to mix the two.

PCRE2 contains a built-in set of character tables that are used by default. These are sufficient for many applications. Normally, the internal tables recognize only ASCII characters. However, when PCRE2 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 built-in tables can be overridden by tables supplied by the application that calls PCRE2. 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 **pcre2_maketables()** function, in the relevant locale. The only argument to this function is a general context, which can be used to pass a custom memory allocator. If the argument is NULL, the system **malloc()** is used. The result can be passed to **pcre2_compile()** as often as necessary, by creating a compile context and calling **pcre2_set_character_tables()** to set the tables pointer therein.

For example, to build and use tables that are appropriate for the French locale (where accented characters with values greater than 127 are treated as letters), the following code could be used:

```
  setlocale(LC_CTYPE, "fr_FR");
  tables = pcre2_maketables(NULL);
  ccontext = pcre2_compile_context_create(NULL);
  pcre2_set_character_tables(ccontext, tables);
  re = pcre2_compile(..., ccontext);
```

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".

The pointer that is passed (via the compile context) to **pcre2_compile()** is saved with the compiled pattern, and the same tables are used by the matching functions. Thus, for any single pattern, compilation and matching both happen in the same locale, but different patterns can be processed in different locales.

It is the caller's responsibility to ensure that the memory containing the tables remains available while they are still in use. When they are no longer needed, you can discard them using **pcre2_maketables_free()**, which should pass as its first parameter the same global context that was used to create the tables.

Saving locale tables

The tables described above are just a sequence of binary bytes, which makes them independent of hardware characteristics such as endianness or whether the processor is 32-bit or 64-bit. A copy of the result of **pcre2_maketables()** can therefore be saved in a file or elsewhere and re-used later, even in a different program or on another computer. The size of the tables (number of bytes) must be obtained by calling **pcre2_config()** with the PCRE2_CONFIG_TABLES_LENGTH option because **pcre2_maketables()** does not return this value. Note that the **pcre2_dftables** program, which is part of the PCRE2 build system, can be used stand-alone to create a file that contains a set of binary tables. See the **pcre2build** documentation for details.

INFORMATION ABOUT A COMPILED PATTERN

int pcre2_pattern_info(const pcre2 ****code***, uint32_t** *what***, void ****where***);**

The **pcre2_pattern_info()** function returns general information about a compiled pattern. For information about callouts, see the next section. The first argument for **pcre2_pattern_info()** is a pointer to the compiled pattern. The second argument specifies which piece of information is required, and the third argument is a pointer to a variable to receive the data. If the third argument is NULL, the first argument is ignored, and the function returns the size in bytes of the variable that is required for the information requested. Otherwise, the yield of the function is zero for success, or one of the following negative numbers:

```
  PCRE2_ERROR_NULL           the argument code was NULL
  PCRE2_ERROR_BADMAGIC       the "magic number" was not found
  PCRE2_ERROR_BADOPTION      the value of what was invalid
  PCRE2_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. Here is a typical call of **pcre2_pattern_info()**, to obtain the length of the compiled pattern:

```
  int rc;
  size_t length;
  rc = pcre2_pattern_info(
    re,               /* result of pcre2_compile() */
    PCRE2_INFO_SIZE,  /* what is required */
    &length);         /* where to put the data */
```

The possible values for the second argument are defined in **pcre2.h**, and are as follows:

```
  PCRE2_INFO_ALLOPTIONS
  PCRE2_INFO_ARGOPTIONS
  PCRE2_INFO_EXTRAOPTIONS
```

Return copies of the pattern's options. The third argument should point to a **uint32_t** variable. PCRE2_INFO_ARGOPTIONS returns exactly the options that were passed to **pcre2_compile()**, whereas PCRE2_INFO_ALLOPTIONS returns the compile options as modified by any top-level (*XXX) option settings such as (*UTF) at the start of the pattern itself. PCRE2_INFO_EXTRAOPTIONS returns the extra options that were set in the compile context by calling the pcre2_set_compile_extra_options() function.

For example, if the pattern /(*UTF)abc/ is compiled with the PCRE2_EXTENDED option, the result for PCRE2_INFO_ALLOPTIONS is PCRE2_EXTENDED and PCRE2_UTF. Option settings such as (?i) that can change within a pattern do not affect the result of PCRE2_INFO_ALLOPTIONS, even if they appear right at the start of the pattern. (This was different in some earlier releases.)

A pattern compiled without PCRE2_ANCHORED is automatically anchored by PCRE2 if the first significant item in every top-level branch is one of the following:

```
  ˆ    unless PCRE2_MULTILINE is set
  \A   always
  \G   always
  .*   sometimes - see below
```

When .* is the first significant item, anchoring is possible only when all the following are true:

```
  .* is not in an atomic group
  .* is not in a capture group that is the subject
       of a backreference
  PCRE2_DOTALL is in force for .*
  Neither (*PRUNE) nor (*SKIP) appears in the pattern
  PCRE2_NO_DOTSTAR_ANCHOR is not set
```

For patterns that are auto-anchored, the PCRE2_ANCHORED bit is set in the options returned for PCRE2_INFO_ALLOPTIONS.

```
  PCRE2_INFO_BACKREFMAX
```

Return the number of the highest backreference in the pattern. The third argument should point to a **uint32_t** variable. Named capture groups acquire numbers as well as names, and these count towards the highest backreference. Backreferences such as \4 or \g{12} match the captured characters of the given group, but in addition, the check that a capture group is set in a conditional group such as (?(3)a|b) is also a backreference. Zero is returned if there are no backreferences.

```
  PCRE2_INFO_BSR
```

The output is a uint32_t integer whose value indicates what character sequences the \R escape sequence matches. A value of PCRE2_BSR_UNICODE means that \R matches any Unicode line ending sequence; a value of PCRE2_BSR_ANYCRLF means that \R matches only CR, LF, or CRLF.

```
  PCRE2_INFO_CAPTURECOUNT
```

Return the highest capture group number in the pattern. In patterns where (?| is not used, this is also the total number of capture groups. The third argument should point to a **uint32_t** variable.

```
  PCRE2_INFO_DEPTHLIMIT
```

If the pattern set a backtracking depth limit by including an item of the form (*LIMIT_DEPTH=nnnn) at the start, the value is returned. The third argument should point to a uint32_t integer. If no such value has been set, the call to **pcre2_pattern_info()** returns the error PCRE2_ERROR_UNSET. Note that this limit will only be used during matching if it is less than the limit set or defaulted by the caller of the match function.

```
  PCRE2_INFO_FIRSTBITMAP
```

In the absence of a single first code unit for a non-anchored pattern, **pcre2_compile()** may construct a 256-bit table that defines a fixed set of values for the first code unit in any match. For example, a pattern that starts with [abc] results in a table with three bits set. When code unit values greater than 255 are supported, the flag bit for 255 means "any code unit of value 255 or above". If such a table was constructed, a pointer to it is returned. Otherwise NULL is returned. The third argument should point to a **const uint8_t *** variable.

```
  PCRE2_INFO_FIRSTCODETYPE
```

Return information about the first code unit of any matched string, for a non-anchored pattern. The third argument should point to a **uint32_t** 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 value can be retrieved using PCRE2_INFO_FIRSTCODEUNIT. If there is no fixed first value, but it is known that a match can occur only at the start of the subject or following a newline in the subject, 2 is returned. Otherwise, and for anchored patterns, 0 is returned.

```
  PCRE2_INFO_FIRSTCODEUNIT
```

Return the value of the first code unit of any matched string for a pattern where PCRE2_INFO_FIRSTCODETYPE returns 1; otherwise return 0. The third argument should point to a **uint32_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.

```
  PCRE2_INFO_FRAMESIZE
```

Return the size (in bytes) of the data frames that are used to remember backtracking positions when the pattern is processed by **pcre2_match()** without the use of JIT. The third argument should point to a **size_t** variable. The frame size depends on the number of capturing parentheses in the pattern. Each additional capture group adds two PCRE2_SIZE variables.

PCRE2_INFO_HASBACKSLASHC

Return 1 if the pattern contains any instances of \C, otherwise 0. The third argument should point to a **uint32_t** variable.

PCRE2_INFO_HASCRORLF

Return 1 if the pattern contains any explicit matches for CR or LF characters, otherwise 0. The third argument should point to a **uint32_t** variable. An explicit match is either a literal CR or LF character, or \r or \n or one of the equivalent hexadecimal or octal escape sequences.

PCRE2_INFO_HEAPLIMIT

If the pattern set a heap memory limit by including an item of the form (*LIMIT_HEAP=nnnn) at the start, the value is returned. The third argument should point to a uint32_t integer. If no such value has been set, the call to **pcre2_pattern_info()** returns the error PCRE2_ERROR_UNSET. Note that this limit will only be used during matching if it is less than the limit set or defaulted by the caller of the match function.

PCRE2_INFO_JCHANGED

Return 1 if the (?J) or (?-J) option setting is used in the pattern, otherwise 0. The third argument should point to a **uint32_t** variable. (?J) and (?-J) set and unset the local PCRE2_DUPNAMES option, respectively.

PCRE2_INFO_JITSIZE

If the compiled pattern was successfully processed by **pcre2_jit_compile()**, return the size of the JIT compiled code, otherwise return zero. The third argument should point to a **size_t** variable.

PCRE2_INFO_LASTCODETYPE

Returns 1 if there is a rightmost literal code unit that must exist in any matched string, other than at its start. The third argument should point to a **uint32_t** variable. If there is no such value, 0 is returned. When 1 is returned, the code unit value itself can be retrieved using PCRE2_INFO_LASTCODEUNIT. 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 1 (with "z" returned from PCRE2_INFO_LASTCODEUNIT), but for /^a\dz\d/ the returned value is 0.

PCRE2_INFO_LASTCODEUNIT

Return the value of the rightmost literal code unit that must exist in any matched string, other than at its start, for a pattern where PCRE2_INFO_LASTCODETYPE returns 1. Otherwise, return 0. The third argument should point to a **uint32_t** variable.

PCRE2_INFO_MATCHEMPTY

Return 1 if the pattern might match an empty string, otherwise 0. The third argument should point to a **uint32_t** variable. When a pattern contains recursive subroutine calls it is not always possible to determine whether or not it can match an empty string. PCRE2 takes a cautious approach and returns 1 in such cases.

PCRE2_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 third argument should point to a uint32_t integer. If no such value has been set, the call to **pcre2_pattern_info()** returns the error PCRE2_ERROR_UNSET. Note that this limit will only be used during matching if it is less than the limit set or defaulted by the caller of the match function.

```
  PCRE2_INFO_MAXLOOKBEHIND
```

A lookbehind assertion moves back a certain number of characters (not code units) when it starts to process each of its branches. This request returns the largest of these backward moves. The third argument should point to a uint32_t integer. The simple assertions \b and \B require a one-character lookbehind and cause PCRE2_INFO_MAXLOOKBEHIND to return 1 in the absence of anything longer. \A also registers a one-character lookbehind, though it does not actually inspect the previous character.

Note that this information is useful for multi-segment matching only if the pattern contains no nested lookbehinds. For example, the pattern (?<=a(?<=ba)c) returns a maximum lookbehind of 2, but when it is processed, the first lookbehind moves back by two characters, matches one character, then the nested lookbehind also moves back by two characters. This puts the matching point three characters earlier than it was at the start. PCRE2_INFO_MAXLOOKBEHIND is really only useful as a debugging tool. See the **pcre2partial** documentation for a discussion of multi-segment matching.

```
  PCRE2_INFO_MINLENGTH
```

If a minimum length for matching subject strings was computed, its value is returned. Otherwise the returned value is 0. This value is not computed when PCRE2_NO_START_OPTIMIZE is set. The value is a number of characters, which in UTF mode may be different from the number of code units. The third argument should point to a **uint32_t** variable. The 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.

```
  PCRE2_INFO_NAMECOUNT
  PCRE2_INFO_NAMEENTRYSIZE
  PCRE2_INFO_NAMETABLE
```

PCRE2 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 **pcre2_substring_get_byname()** 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 **pcre2_match()** 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. PCRE2_INFO_NAMECOUNT gives the number of entries, and PCRE2_INFO_NAMEENTRYSIZE gives the size of each entry in code units; both of these return a **uint32_t** value. The entry size depends on the length of the longest name.

PCRE2_INFO_NAMETABLE returns a pointer to the first entry of the table. This is a PCRE2_SPTR pointer to a block of code units. In the 8-bit library, 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 code units, the first of which contains the parenthesis number. In the 32-bit library, the pointer points to 32-bit code 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 capture groups with the same number, as described in the section on duplicate group numbers in the **pcre2pattern** 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 capture groups with different numbers are permitted, but only if PCRE2_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 capture groups 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 PCRE2_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 capture groups, 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 capture groups using the name-to-number map, remember that the length of the entries is likely to be different for each compiled pattern.

```
  PCRE2_INFO_NEWLINE
```

The output is one of the following **uint32_t** values:

```
  PCRE2_NEWLINE_CR       Carriage return (CR)
  PCRE2_NEWLINE_LF       Linefeed (LF)
  PCRE2_NEWLINE_CRLF     Carriage return, linefeed (CRLF)
  PCRE2_NEWLINE_ANY      Any Unicode line ending
  PCRE2_NEWLINE_ANYCRLF  Any of CR, LF, or CRLF
  PCRE2_NEWLINE_NUL      The NUL character (binary zero)
```

This identifies the character sequence that will be recognized as meaning "newline" while matching.

```
  PCRE2_INFO_SIZE
```

Return the size of the compiled pattern in bytes (for all three libraries). The third argument should point to a **size_t** variable. This value includes the size of the general data block that precedes the code units of the compiled pattern itself. The value that is used when **pcre2_compile()** is getting memory in which to place the compiled pattern may be slightly larger than the value returned by this option, because there are cases where the code that calculates the size has to over-estimate. Processing a pattern with the JIT compiler does not alter the value returned by this option.

INFORMATION ABOUT A PATTERN'S CALLOUTS

**int pcre2_callout_enumerate(const pcre2_code ****code*,
  int (**callback*)**(pcre2_callout_enumerate_block *, void *),**
  **void ****user_data*);

A script language that supports the use of string arguments in callouts might like to scan all the callouts in a pattern before running the match. This can be done by calling **pcre2_callout_enumerate()**. The first argument is a pointer to a compiled pattern, the second points to a callback function, and the third is arbitrary user data. The callback function is called for every callout in the pattern in the order in which they appear. Its first argument is a pointer to a callout enumeration block, and its second argument is the *user_data* value that was passed to **pcre2_callout_enumerate()**. The contents of the callout enumeration block are described in the **pcre2callout** documentation, which also gives further details about callouts.

SERIALIZATION AND PRECOMPILING

It is possible to save compiled patterns on disc or elsewhere, and reload them later, subject to a number of restrictions. The host on which the patterns are reloaded must be running the same version of PCRE2, with the same code unit width, and must also have the same endianness, pointer width, and PCRE2_SIZE type. Before compiled patterns can be saved, they must be converted to a "serialized" form, which in the case of PCRE2 is really just a bytecode dump. The functions whose names begin with **pcre2_serialize_** are used for converting to and from the serialized form. They are described in the **pcre2serialize** documentation. Note that PCRE2 serialization does not convert compiled patterns to an abstract format like Java or .NET serialization.

THE MATCH DATA BLOCK

pcre2_match_data *pcre2_match_data_create(uint32_t *ovecsize*,
  pcre2_general_context ***gcontext*);**

pcre2_match_data *pcre2_match_data_create_from_pattern(
  const pcre2_code ***code*, **pcre2_general_context ***gcontext*);**

void pcre2_match_data_free(pcre2_match_data ***match_data*);**

Information about a successful or unsuccessful match is placed in a match data block, which is an opaque structure that is accessed by function calls. In particular, the match data block contains a vector of offsets into the subject string that define the matched parts of the subject. This is known as the *ovector*.

Before calling **pcre2_match()**, **pcre2_dfa_match()**, or **pcre2_jit_match()** you must create a match data block by calling one of the creation functions above. For **pcre2_match_data_create()**, the first argument is the number of pairs of offsets in the *ovector*.

When using **pcre2_match()**, one pair of offsets is required to identify the string that matched the whole pattern, with an additional pair for each captured substring. For example, a value of 4 creates enough space to record the matched portion of the subject plus three captured substrings.

When using **pcre2_dfa_match()** there may be multiple matched substrings of different lengths at the same point in the subject. The ovector should be made large enough to hold as many as are expected.

A minimum of at least 1 pair is imposed by **pcre2_match_data_create()**, so it is always possible to return the overall matched string in the case of **pcre2_match()** or the longest match in the case of **pcre2_dfa_match()**.

The second argument of **pcre2_match_data_create()** is a pointer to a general context, which can specify custom memory management for obtaining the memory for the match data block. If you are not using custom memory management, pass NULL, which causes **malloc()** to be used.

For **pcre2_match_data_create_from_pattern()**, the first argument is a pointer to a compiled pattern. The ovector is created to be exactly the right size to hold all the substrings a pattern might capture when matched using **pcre2_match()**. You should not use this call when matching with **pcre2_dfa_match()**. The second argument is again a pointer to a general context, but in this case if NULL is passed, the memory is obtained using the same allocator that was used for the compiled pattern (custom or default).

A match data block can be used many times, with the same or different compiled patterns. You can extract information from a match data block after a match operation has finished, using functions that are described in the sections on matched strings and other match data below.

When a call of **pcre2_match()** fails, valid data is available in the match block only when the error is PCRE2_ERROR_NOMATCH, PCRE2_ERROR_PARTIAL, or one of the error codes for an invalid UTF string. Exactly what is available depends on the error, and is detailed below.

When one of the matching functions is called, pointers to the compiled pattern and the subject string are set in the match data block so that they can be referenced by the extraction functions after a successful match. After running a match, you must not free a compiled pattern or a subject string until after all operations on the match data block (for that match) have taken place, unless, in the case of the subject string, you have used the PCRE2_COPY_MATCHED_SUBJECT option, which is described in the section entitled "Option bits for **pcre2_match()**" below.

When a match data block itself is no longer needed, it should be freed by calling **pcre2_match_data_free()**. If this function is called with a NULL argument, it returns immediately, without doing anything.

MATCHING A PATTERN: THE TRADITIONAL FUNCTION

int pcre2_match(const pcre2_code ****code*, PCRE2_SPTR** *subject***,**
  PCRE2_SIZE *length***, PCRE2_SIZE** *startoffset***,**
  uint32_t *options***, pcre2_match_data ****match_data***,**
  pcre2_match_context ****mcontext*);**

The function **pcre2_match()** is called to match a subject string against a compiled pattern, which is passed in the *code* argument. You can call **pcre2_match()** with the same *code* argument as many times as you like, in order to find multiple matches in the subject string or 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 **pcre2_dfa_match()** function.

Here is an example of a simple call to **pcre2_match()**:

```
  pcre2_match_data *md = pcre2_match_data_create(4, NULL);
  int rc = pcre2_match(
    re,             /* result of pcre2_compile() */
    "some string",  /* the subject string */
    11,             /* the length of the subject string */
    0,              /* start at offset 0 in the subject */
    0,              /* default options */
    md,             /* the match data block */
    NULL);          /* a match context; NULL means use defaults */
```

If the subject string is zero-terminated, the length can be given as PCRE2_ZERO_TERMINATED. A match context must be provided if certain less common matching parameters are to be changed. For details, see the section on the match context above.

The string to be matched by pcre2_match()

The subject string is passed to **pcre2_match()** as a pointer in *subject*, a length in *length*, and a starting offset in *startoffset*. The length and offset are in code units, not characters. That is, they are in bytes for the 8-bit library, 16-bit code units for the 16-bit library, and 32-bit code units for the 32-bit library, whether or not UTF processing is enabled.

If *startoffset* is greater than the length of the subject, **pcre2_match()** returns PCRE2_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 starting offset must point to the start of a character, or to the end of the subject (in UTF-32 mode, one code unit equals one character, so all offsets are valid). Like the pattern string, the subject may contain binary zeros.

A non-zero starting offset is useful when searching for another match in the same subject by calling **pcre2_match()** again after a previous success. Setting *startoffset* differs from passing over a shortened string and setting PCRE2_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 **pcre2_match()** finds the first occurrence. If **pcre2_match()** 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 **pcre2_match()** 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 PCRE2_NOTEMPTY_ATSTART and PCRE2_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 **pcre2demo** 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, a single 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. In other words, the anchoring must be the result of setting the PCRE2_ANCHORED option or the use of .* with PCRE2_DOTALL, not by starting the pattern with ˆ or \A.

Option bits for pcre2_match()

The unused bits of the *options* argument for **pcre2_match()** must be zero. The only bits that may be set are PCRE2_ANCHORED, PCRE2_COPY_MATCHED_SUBJECT, PCRE2_ENDANCHORED, PCRE2_NOTBOL, PCRE2_NOTEOL, PCRE2_NOTEMPTY, PCRE2_NOTEMPTY_ATSTART, PCRE2_NO_JIT, PCRE2_NO_UTF_CHECK, PCRE2_PARTIAL_HARD, and PCRE2_PARTIAL_SOFT. Their action is described below.

Setting PCRE2_ANCHORED or PCRE2_ENDANCHORED at match time is not supported by the just-in-time (JIT) compiler. If it is set, JIT matching is disabled and the interpretive code in **pcre2_match()** is run. Apart from PCRE2_NO_JIT (obviously), the remaining options are supported for JIT matching.

```
  PCRE2_ANCHORED
```

The PCRE2_ANCHORED option limits **pcre2_match()** to matching at the first matching position. If a pattern was compiled with PCRE2_ANCHORED, or turned out to be anchored by virtue of its contents, it cannot be made unachored at matching time. Note that setting the option at match time disables JIT matching.

```
  PCRE2_COPY_MATCHED_SUBJECT
```

By default, a pointer to the subject is remembered in the match data block so that, after a successful match, it can be referenced by the substring extraction functions. This means that the subject's memory must not be freed until all such operations are complete. For some applications where the lifetime of the subject string is not guaranteed, it may be necessary to make a copy of the subject string, but it is wasteful to do this unless the match is successful. After a successful match, if PCRE2_COPY_MATCHED_SUBJECT is set, the subject is copied and the new pointer is remembered in the match data block instead of the original subject pointer. The memory allocator that was used for the match block itself is used. The copy is automatically freed when **pcre2_match_data_free()** is called to free the match data block. It is also automatically freed if the match data block is re-used for another match operation.

```
  PCRE2_ENDANCHORED
```

If the PCRE2_ENDANCHORED option is set, any string that **pcre2_match()** matches must be right at the end of the subject string. Note that setting the option at match time disables JIT matching.

```
  PCRE2_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 having set PCRE2_MULTILINE at compile time causes circumflex never to match. This option affects only the behaviour of the circumflex metacharacter. It does not affect \A.

```
  PCRE2_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 having set PCRE2_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.

```
  PCRE2_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 PCRE2_NOTEMPTY set, this match is not valid, so **pcre2_match()** searches further into the string for occurrences of "a" or "b".

```
  PCRE2_NOTEMPTY_ATSTART
```

This is like PCRE2_NOTEMPTY, except that it locks out an empty string match only at the first matching position, that is, at the start of the subject plus the starting offset. An empty string match later in the subject is permitted. If the pattern is anchored, such a match can occur only if the pattern contains \K.

```
  PCRE2_NO_JIT
```

By default, if a pattern has been successfully processed by **pcre2_jit_compile()**, JIT is automatically used when **pcre2_match()** is called with options that JIT supports. Setting PCRE2_NO_JIT disables the use of JIT; it forces matching to be done by the interpreter.

```
  PCRE2_NO_UTF_CHECK
```

When PCRE2_UTF is set at compile time, the validity of the subject as a UTF string is checked unless PCRE2_NO_UTF_CHECK is passed to **pcre2_match()** or PCRE2_MATCH_INVALID_UTF was passed to **pcre2_compile()**. The latter special case is discussed in detail in the **pcre2unicode** documentation.

In the default case, if a non-zero starting offset is given, the check is applied only to that part of the subject that could be inspected during matching, and there is a check that the starting offset points to the first code unit of a character or to the end of the subject. If there are no lookbehind assertions in the pattern, the check starts at the starting offset. Otherwise, it starts at the length of the longest lookbehind before the starting offset, or at the start of the subject if there are not that many characters before the starting offset. Note that the sequences \b and \B are one-character lookbehinds.

The check is carried out before any other processing takes place, and a negative error code is returned if the check fails. There are several UTF error codes for each code unit width, corresponding to different problems with the code unit sequence. There are discussions about the validity of UTF-8 strings, UTF-16 strings, and UTF-32 strings in the **pcre2unicode** documentation.

If you know that your subject is valid, and you want to skip this check for performance reasons, you can set the PCRE2_NO_UTF_CHECK option when calling **pcre2_match()**. You might want to do this for the second and subsequent calls to **pcre2_match()** if you are making repeated calls to find multiple matches in the same subject string.

Warning: Unless PCRE2_MATCH_INVALID_UTF was set at compile time, when PCRE2_NO_UTF_CHECK is set at match time the effect of passing an invalid string as a subject, or an invalid value of *startoffset*, is undefined. Your program may crash or loop indefinitely or give wrong results.

```
PCRE2_PARTIAL_HARD
PCRE2_PARTIAL_SOFT
```

These options turn on the partial matching feature. 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. In addition, either at least one character must have been inspected or the pattern must contain a lookbehind, or the pattern must be one that could match an empty string.

If this situation arises when PCRE2_PARTIAL_SOFT (but not PCRE2_PARTIAL_HARD) is set, matching continues by testing any remaining alternatives. Only if no complete match can be found is PCRE2_ERROR_PARTIAL returned instead of PCRE2_ERROR_NOMATCH. In other words, PCRE2_PARTIAL_SOFT specifies that the caller is prepared to handle a partial match, but only if no complete match can be found.

If PCRE2_PARTIAL_HARD is set, it overrides PCRE2_PARTIAL_SOFT. In this case, if a partial match is found, **pcre2_match()** immediately returns PCRE2_ERROR_PARTIAL, without considering any other alternatives. In other words, when PCRE2_PARTIAL_HARD is set, a partial match is considered to be more important that an alternative complete match.

There is a more detailed discussion of partial and multi-segment matching, with examples, in the **pcre2partial** documentation.

NEWLINE HANDLING WHEN MATCHING

When PCRE2 is built, a default newline convention is set; this is usually the standard convention for the operating system. The default can be overridden in a compile context by calling **pcre2_set_newline()**. It can also be overridden by starting a pattern string with, for example, (*CRLF), as described in the section on newline conventions in the **pcre2pattern** page. During matching, the newline choice affects the behaviour of the dot, circumflex, and dollar metacharacters. It may also alter the way the match starting position is advanced after a match failure for an unanchored pattern.

When PCRE2_NEWLINE_CRLF, PCRE2_NEWLINE_ANYCRLF, or PCRE2_NEWLINE_ANY is set as the newline convention, and a match attempt for an unanchored pattern fails when the current starting 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 PCRE2_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 in the pattern, or one of the \r or \n or equivalent octal or hexadecimal escape sequences. Implicit matches such as [^X] do not count, nor does \s, even though it 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.

HOW PCRE2_MATCH() RETURNS A STRING AND CAPTURED SUBSTRINGS

uint32_t pcre2_get_ovector_count(pcre2_match_data ****match_data*);**

PCRE2_SIZE *pcre2_get_ovector_pointer(pcre2_match_data ***match_data*);**

In general, a pattern matches a certain portion of the subject, and in addition, further substrings from the subject may be picked out by parenthesized parts of the pattern. Following the usage in Jeffrey Friedl's book, this is called "capturing" in what follows, and the phrase "capture group" (Perl terminology) is used for a fragment of a pattern that picks out a substring. PCRE2 supports several other kinds of parenthesized group that do not cause substrings to be captured. The **pcre2_pattern_info()** function can be used to find out how many capture groups there are in a compiled pattern.

You can use auxiliary functions for accessing captured substrings by number or by name, as described in sections below.

Alternatively, you can make direct use of the vector of PCRE2_SIZE values, called the **ovector**, which contains the offsets of captured strings. It is part of the match data block. The function **pcre2_get_ovector_pointer()** returns the address of the ovector, and **pcre2_get_ovector_count()** returns the number of pairs of values it contains.

Within the ovector, the first in each pair of values is set to the offset of the first code unit of a substring, and the second is set to the offset of the first code unit after the end of a substring. These values are always code unit offsets, not character offsets. That is, they are byte offsets in the 8-bit library, 16-bit offsets in the 16-bit library, and 32-bit offsets in the 32-bit library.

After a partial match (error return PCRE2_ERROR_PARTIAL), only the first pair of offsets (that is, *ovector[0]* and *ovector[1]*) are set. They identify the part of the subject that was partially matched. See the **pcre2partial** documentation for details of partial matching.

After a fully successful match, the first pair of offsets identifies the portion of the subject string that was matched by the entire pattern. The next pair is used for the first captured substring, and so on. The value returned by **pcre2_match()** 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 captured substrings, the return value from a successful match is 1, indicating that just the first pair of offsets has been set.

If a pattern uses the \K escape sequence within a positive assertion, the reported start of a successful match can be greater than the end of the match. For example, if the pattern (?=ab\K) is matched against "ab", the start and end offset values for the match are 2 and 0.

If a capture group is matched repeatedly within a single match operation, it is the last portion of the subject that it matched that is returned.

If the ovector is too small to hold all the captured substring offsets, as much as possible is filled in, and the function returns a value of zero. If captured substrings are not of interest, **pcre2_match()** may be called with a match data block whose ovector is of minimum length (that is, one pair).

It is possible for capture group number *n+1* to match some part of the subject when group *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 groups 1 and 3 are matched, but 2 is not. When this happens, both values in the offset pairs corresponding to unused groups are set to PCRE2_UNSET.

Offset values that correspond to unused groups at the end of the expression are also set to PCRE2_UNSET. For example, if the string "abc" is matched against the pattern (abc)(x(yz)?)? groups 2 and 3 are not matched. The return from the function is 2, because the highest used capture group number is 1. The offsets for for the second and third capture groupss (assuming the vector is large enough, of course) are set to PCRE2_UNSET.

Elements in the 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 **pcre2_match()**. The other elements retain whatever values they previously had. After a failed match attempt, the contents of the ovector are unchanged.

OTHER INFORMATION ABOUT A MATCH

PCRE2_SPTR pcre2_get_mark(pcre2_match_data ***match_data*);**

PCRE2_SIZE pcre2_get_startchar(pcre2_match_data ****match_data*);**

As well as the offsets in the ovector, other information about a match is retained in the match data block and can be retrieved by the above functions in appropriate circumstances. If they are called at other times, the result is undefined.

After a successful match, a partial match (PCRE2_ERROR_PARTIAL), or a failure to match (PCRE2_ERROR_NOMATCH), a mark name may be available. The function **pcre2_get_mark()** can be called to access this name, which can be specified in the pattern by any of the backtracking control verbs, not just (*MARK). The same function applies to all the verbs. It returns a pointer to the zero-terminated name, which is within the compiled pattern. If no name is available, NULL is returned. The length of the name (excluding the terminating zero) is stored in the code unit that precedes the name. You should use this length instead of relying on the terminating zero if the name might contain a binary zero.

After a successful match, the name that is returned is the last mark name encountered on the matching path through the pattern. Instances of backtracking verbs without names do not count. Thus, for example, if the matching path contains (*MARK:A)(*PRUNE), the name "A" is returned. After a "no match" or a partial match, the last encountered name is returned. For example, consider this pattern:

```
  ^(*MARK:A)((*MARK:B)a|b)c
```

When it matches "bc", the returned name is A. The B mark is "seen" in the first branch of the group, but it is not on the matching path. On the other hand, when this pattern fails to match "bx", the returned name is B.

Warning: By default, certain start-of-match optimizations are used to give a fast "no match" result in some situations. For example, if the anchoring is removed from the pattern above, there is an initial check for the presence of "c" in the subject before running the matching engine. This check fails for "bx", causing a match failure without seeing any marks. You can disable the start-of-match optimizations by setting the PCRE2_NO_START_OPTIMIZE option for **pcre2_compile()** or by starting the pattern with (*NO_START_OPT).

After a successful match, a partial match, or one of the invalid UTF errors (for example, PCRE2_ERROR_UTF8_ERR5), **pcre2_get_startchar()** can be called. After a successful or partial match it returns the code unit offset of the character at which the match started. For a non-partial match, this can be different to the value of *ovector[0]* if the pattern contains the \K escape sequence. After a partial match, however, this value is always the same as *ovector[0]* because \K does not affect the result of a partial match.

After a UTF check failure, **pcre2_get_startchar()** can be used to obtain the code unit offset of the invalid UTF character. Details are given in the **pcre2unicode** page.

ERROR RETURNS FROM pcre2_match()

If **pcre2_match()** fails, it returns a negative number. This can be converted to a text string by calling the **pcre2_get_error_message()** function (see "Obtaining a textual error message" below). Negative error codes are also returned by other functions, and are documented with them. The codes are given names in the header file. If UTF checking is in force and an invalid UTF subject string is detected, one of a number of UTF-specific negative error codes is returned. Details are given in the **pcre2unicode** page. The following are the other errors that may be returned by **pcre2_match()**:

```
  PCRE2_ERROR_NOMATCH
```

The subject string did not match the pattern.

```
  PCRE2_ERROR_PARTIAL
```

The subject string did not match, but it did match partially. See the **pcre2partial** documentation for details of partial matching.

```
  PCRE2_ERROR_BADMAGIC
```

PCRE2 stores a 4-byte "magic number" at the start of the compiled code, to catch the case when it is passed a junk pointer. This is the error that is returned when the magic number is not present.

```
  PCRE2_ERROR_BADMODE
```

This error is given when a compiled pattern is passed to a function in a library of a different code unit width, for example, a pattern compiled by the 8-bit library is passed to a 16-bit or 32-bit library function.

```
  PCRE2_ERROR_BADOFFSET
```

The value of *startoffset* was greater than the length of the subject.

```
  PCRE2_ERROR_BADOPTION
```

An unrecognized bit was set in the *options* argument.

```
  PCRE2_ERROR_BADUTFOFFSET
```

The UTF code unit sequence that was passed as a subject was checked and found to be valid (the PCRE2_NO_UTF_CHECK option was not set), but the value of *startoffset* did not point to the beginning of a UTF character or the end of the subject.

```
  PCRE2_ERROR_CALLOUT
```

This error is never generated by **pcre2_match()** itself. It is provided for use by callout functions that want to cause **pcre2_match()** or **pcre2_callout_enumerate()** to return a distinctive error code. See the **pcre2callout** documentation for details.

```
  PCRE2_ERROR_DEPTHLIMIT
```

The nested backtracking depth limit was reached.

```
  PCRE2_ERROR_HEAPLIMIT
```

The heap limit was reached.

```
  PCRE2_ERROR_INTERNAL
```

An unexpected internal error has occurred. This error could be caused by a bug in PCRE2 or by overwriting of the compiled pattern.

```
  PCRE2_ERROR_JIT_STACKLIMIT
```

This error is returned when a pattern that was successfully studied using JIT is being matched, but the memory available for the just-in-time processing stack is not large enough. See the **pcre2jit** documentation for more details.

```
  PCRE2_ERROR_MATCHLIMIT
```

The backtracking match limit was reached.

```
  PCRE2_ERROR_NOMEMORY
```

If a pattern contains many nested backtracking points, heap memory is used to remember them. This error is given when the memory allocation function (default or custom) fails. Note that a different error, PCRE2_ERROR_HEAPLIMIT, is given if the amount of memory needed exceeds the heap limit. PCRE2_ERROR_NOMEMORY is also returned if PCRE2_COPY_MATCHED_SUBJECT is set and memory allocation fails.

```
  PCRE2_ERROR_NULL
```

Either the *code*, *subject*, or *match_data* argument was passed as NULL.

```
  PCRE2_ERROR_RECURSELOOP
```

This error is returned when **pcre2_match()** detects a recursion loop within the pattern. Specifically, it means that either the whole pattern or a capture group 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 groups, cannot be detected until matching is attempted.

OBTAINING A TEXTUAL ERROR MESSAGE

int pcre2_get_error_message(int *errorcode***, PCRE2_UCHAR ***buffer***,**
  PCRE2_SIZE *bufflen***);**

A text message for an error code from any PCRE2 function (compile, match, or auxiliary) can be obtained by calling **pcre2_get_error_message()**. The code is passed as the first argument, with the remaining two arguments specifying a code unit buffer and its length in code units, into which the text message is placed. The message is returned in code units of the appropriate width for the library that is being used.

The returned message is terminated with a trailing zero, and the function returns the number of code units used, excluding the trailing zero. If the error number is unknown, the negative error code PCRE2_ERROR_BADDATA is returned. If the buffer is too small, the message is truncated (but still with a trailing zero), and the negative error code PCRE2_ERROR_NOMEMORY is returned. None of the messages are very long; a buffer size of 120 code units is ample.

EXTRACTING CAPTURED SUBSTRINGS BY NUMBER

int pcre2_substring_length_bynumber(pcre2_match_data ***match_data*,**
  uint32_t *number***, PCRE2_SIZE ***length***);**

int pcre2_substring_copy_bynumber(pcre2_match_data ***match_data*,**
  uint32_t *number***, PCRE2_UCHAR ***buffer***,**
  PCRE2_SIZE ***bufflen*);**

int pcre2_substring_get_bynumber(pcre2_match_data ***match_data*,**
  uint32_t *number***, PCRE2_UCHAR ****bufferptr***,**
  PCRE2_SIZE ***bufflen*);**

void pcre2_substring_free(PCRE2_UCHAR ***buffer*);**

Captured substrings can be accessed directly by using the ovector as described above. For convenience, auxiliary functions are provided for extracting captured substrings as new, separate, zero-terminated strings. 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.

The functions in this section identify substrings by number. The number zero refers to the entire matched substring, with higher numbers referring to substrings captured by parenthesized groups. After a partial match, only substring zero is available. An attempt to extract any other substring gives the error PCRE2_ERROR_PARTIAL. The next section describes similar functions for extracting captured substrings by name.

If a pattern uses the \K escape sequence within a positive assertion, the reported start of a successful match can be greater than the end of the match. For example, if the pattern (?=ab\K) is matched against "ab", the start and end offset values for the match are 2 and 0. In this situation, calling these functions with a zero substring number extracts a zero-length empty string.

You can find the length in code units of a captured substring without extracting it by calling **pcre2_substring_length_bynumber()**. The first argument is a pointer to the match data block, the second is the group number, and the third is a pointer to a variable into which the length is placed. If you just want to know whether or not the substring has been captured, you can pass the third argument as NULL.

The **pcre2_substring_copy_bynumber()** function copies a captured substring into a supplied buffer, whereas **pcre2_substring_get_bynumber()** copies it into new memory, obtained using the same memory allocation function that was used for the match data block. The first two arguments of these functions are a pointer to the match data block and a capture group number.

The final arguments of **pcre2_substring_copy_bynumber()** are a pointer to the buffer and a pointer to a variable that contains its length in code units. This is updated to contain the actual number of code units used for the extracted substring, excluding the terminating zero.

For **pcre2_substring_get_bynumber()** the third and fourth arguments point to variables that are updated with a pointer to the new memory and the number of code units that comprise the substring, again excluding the terminating zero. When the substring is no longer needed, the memory should be freed by calling **pcre2_substring_free()**.

The return value from all these functions is zero for success, or a negative error code. If the pattern match failed, the match failure code is returned. If a substring number greater than zero is used after a partial match, PCRE2_ERROR_PARTIAL is returned. Other possible error codes are:

```
  PCRE2_ERROR_NOMEMORY
```

The buffer was too small for **pcre2_substring_copy_bynumber()**, or the attempt to get memory failed for **pcre2_substring_get_bynumber()**.

```
  PCRE2_ERROR_NOSUBSTRING
```

There is no substring with that number in the pattern, that is, the number is greater than the number of capturing parentheses.

```
  PCRE2_ERROR_UNAVAILABLE
```

The substring number, though not greater than the number of captures in the pattern, is greater than the number of slots in the ovector, so the substring could not be captured.

```
  PCRE2_ERROR_UNSET
```

The substring did not participate in the match. For example, if the pattern is (abc)|(def) and the subject is "def", and the ovector contains at least two capturing slots, substring number 1 is unset.

EXTRACTING A LIST OF ALL CAPTURED SUBSTRINGS

int pcre2_substring_list_get(pcre2_match_data ****match_data*,**
  PCRE2_UCHAR ******listptr*, PCRE2_SIZE *****lengthsptr***);**

void pcre2_substring_list_free(PCRE2_SPTR ****list*);**

The **pcre2_substring_list_get()** function extracts all available substrings and builds a list of pointers to them. It also (optionally) builds a second list that contains their lengths (in code units), excluding a terminating zero that is added to each of them. All this is done in a single block of memory that is obtained using the same memory allocation function that was used to get the match data block.

This function must be called only after a successful match. If called after a partial match, the error code PCRE2_ERROR_PARTIAL is returned.

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 address of the list of lengths is returned via *lengthsptr*. If your strings do not contain binary zeros and you do not therefore need the lengths, you may supply NULL as the **lengthsptr** argument to disable the creation of a list of lengths. The yield of the function is zero if all went well, or PCRE2_ERROR_NOMEMORY if the memory block could not be obtained. When the list is no longer needed, it should be freed by calling **pcre2_substring_list_free()**.

If this function encounters a substring that is unset, which can happen when capture group number *n+1* matches some part of the subject, but group *n* has not been used at all, it returns an empty string. This can be distinguished from a genuine zero-length substring by inspecting the appropriate offset in the ovector, which contain PCRE2_UNSET for unset substrings, or by calling **pcre2_substring_length_bynumber()**.

EXTRACTING CAPTURED SUBSTRINGS BY NAME

int pcre2_substring_number_from_name(const pcre2_code ****code*,**
  PCRE2_SPTR *name***);**

int pcre2_substring_length_byname(pcre2_match_data ****match_data*,**
  PCRE2_SPTR *name***, PCRE2_SIZE ****length***);**

int pcre2_substring_copy_byname(pcre2_match_data ****match_data*,**
  PCRE2_SPTR *name***, PCRE2_UCHAR ****buffer***, PCRE2_SIZE ****bufflen***);**

int pcre2_substring_get_byname(pcre2_match_data ****match_data*,**
  PCRE2_SPTR *name***, PCRE2_UCHAR *****bufferptr***, PCRE2_SIZE ****bufflen***);**

void pcre2_substring_free(PCRE2_UCHAR ****buffer*);**

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 capture group called "xxx" is 2. If the name is known to be unique (PCRE2_DUPNAMES was not set), you can find the number from the name by calling **pcre2_substring_number_from_name()**. The first argument is the compiled pattern, and the second is the name. The yield of the function is the group number, PCRE2_ERROR_NOSUBSTRING if there is no group with that name, or PCRE2_ERROR_NOUNIQUESUBSTRING if there is more than one group with that name. Given the number, you can extract the substring directly from the ovector, or use one of the "bynumber" functions described above.

For convenience, there are also "byname" functions that correspond to the "bynumber" functions, the only difference being that the second argument is a name instead of a number. If PCRE2_DUPNAMES is set and there are duplicate names, these functions scan all the groups with the given name, and return the captured substring from the first named group that is set.

If there are no groups with the given name, PCRE2_ERROR_NOSUBSTRING is returned. If all groups with the name have numbers that are greater than the number of slots in the ovector, PCRE2_ERROR_UNAVAILABLE is returned. If there is at least one group with a slot in the ovector, but no group is found to be set, PCRE2_ERROR_UNSET is returned.

Warning: If the pattern uses the (?| feature to set up multiple capture groups with the same number, as described in the section on duplicate group numbers in the **pcre2pattern** page, you cannot use names to distinguish the different capture groups, because names are not included in the compiled code. The matching process uses only numbers. For this reason, the use of different names for groups with the same number causes an error at compile time.

CREATING A NEW STRING WITH SUBSTITUTIONS

int pcre2_substitute(const pcre2_code ****code*, PCRE2_SPTR** *subject***,**
PCRE2_SIZE *length***, PCRE2_SIZE** *startoffset***,**
uint32_t *options***, pcre2_match_data ****match_data***,**
pcre2_match_context ****mcontext*, PCRE2_SPTR** *replacement***,**
PCRE2_SIZE *rlength***, PCRE2_UCHAR ****outputbuffer***,**
PCRE2_SIZE ****outlengthptr*);**

This function optionally calls **pcre2_match()** and then makes a copy of the subject string in *outputbuffer*, replacing parts that were matched with the *replacement* string, whose length is supplied in **rlength**. This can be given as PCRE2_ZERO_TERMINATED for a zero-terminated string. There is an option (see PCRE2_SUBSTITUTE_REPLACEMENT_ONLY below) to return just the replacement string(s). The default action is to perform just one replacement if the pattern matches, but there is an option that requests multiple replacements (see PCRE2_SUBSTITUTE_GLOBAL below).

If successful, **pcre2_substitute()** returns the number of substitutions that were carried out. This may be zero if no match was found, and is never greater than one unless PCRE2_SUBSTITUTE_GLOBAL is set. A negative value is returned if an error is detected.

Matches in which a \K item in a lookahead in the pattern causes the match to end before it starts are not supported, and give rise to an error return. For global replacements, matches in which \K in a lookbehind causes the match to start earlier than the point that was reached in the previous iteration are also not supported.

The first seven arguments of **pcre2_substitute()** are the same as for **pcre2_match()**, except that the partial matching options are not permitted, and *match_data* may be passed as NULL, in which case a match data block is obtained and freed within this function, using memory management functions from the match context, if provided, or else those that were used to allocate memory for the compiled code.

If *match_data* is not NULL and PCRE2_SUBSTITUTE_MATCHED is not set, the provided block is used for all calls to **pcre2_match()**, and its contents afterwards are the result of the final call. For global changes, this will always be a no-match error. The contents of the ovector within the match data block may or may not have been changed.

As well as the usual options for **pcre2_match()**, a number of additional options can be set in the *options* argument of **pcre2_substitute()**. One such option is PCRE2_SUBSTITUTE_MATCHED. When this is set, an external *match_data* block must be provided, and it must have been used for an external call to **pcre2_match()**. The data in the *match_data* block (return code, offset vector) is used for the first substitution instead of calling **pcre2_match()** from within **pcre2_substitute()**. This allows an application to check for a match before choosing to substitute, without having to repeat the match.

The contents of the externally supplied match data block are not changed when PCRE2_SUBSTITUTE_MATCHED is set. If PCRE2_SUBSTITUTE_GLOBAL is also set, **pcre2_match()** is called after the first substitution to check for further matches, but this is done using an internally obtained match data block, thus always leaving the external block unchanged.

The *code* argument is not used for matching before the first substitution when PCRE2_SUBSTITUTE_MATCHED is set, but it must be provided, even when PCRE2_SUBSTITUTE_GLOBAL is not set, because it contains information such as the UTF setting and the number of capturing parentheses in the pattern.

The default action of **pcre2_substitute()** is to return a copy of the subject string with matched substrings replaced. However, if PCRE2_SUBSTITUTE_REPLACEMENT_ONLY is set, only the replacement substrings are returned. In the global case, multiple replacements are concatenated in the output buffer. Substitution callouts (see below) can be used to separate them if necessary.

The *outlengthptr* argument of **pcre2_substitute()** must point to a variable that contains the length, in code units, of the output buffer. If the function is successful, the value is updated to contain the length in code units of the new string, excluding the trailing zero that is automatically added.

If the function is not successful, the value set via *outlengthptr* depends on the type of error. For syntax errors in the replacement string, the value is the offset in the replacement string where the error was detected. For other errors, the value is PCRE2_UNSET by default. This includes the case of the output buffer being too small, unless PCRE2_SUBSTITUTE_OVERFLOW_LENGTH is set.

PCRE2_SUBSTITUTE_OVERFLOW_LENGTH changes what happens when the output buffer is too small. The default action is to return PCRE2_ERROR_NOMEMORY immediately. If this option is set, however, **pcre2_substitute()** continues to go through the motions of matching and substituting (without, of course, writing anything) in order to compute the size of buffer that is needed. This value is passed back via the *outlengthptr* variable, with the result of the function still being PCRE2_ERROR_NOMEMORY.

Passing a buffer size of zero is a permitted way of finding out how much memory is needed for given substitution. However, this does mean that the entire operation is carried out twice. Depending on the application, it may be more efficient to allocate a large buffer and free the excess afterwards, instead of using PCRE2_SUBSTITUTE_OVERFLOW_LENGTH.

The replacement string, which is interpreted as a UTF string in UTF mode, is checked for UTF validity unless PCRE2_NO_UTF_CHECK is set. An invalid UTF replacement string causes an immediate return with the relevant UTF error code.

If PCRE2_SUBSTITUTE_LITERAL is set, the replacement string is not interpreted in any way. By default, however, a dollar character is an escape character that can specify the insertion of characters from capture groups and names from (*MARK) or other control verbs in the pattern. The following forms are always recognized:

```
  $$                  insert a dollar character
  $<n> or ${<n>}      insert the contents of group <n>
  $*MARK or ${*MARK}  insert a control verb name
```

Either a group number or a group name can be given for <n>. Curly brackets are required only if the following character would be interpreted as part of the number or name. The number may be zero to include the entire matched string. For example, if the pattern a(b)c is matched with "=abc=" and the replacement string "+$1$0$1+", the result is "=+babcb+=".

$*MARK inserts the name from the last encountered backtracking control verb on the matching path that has a name. (*MARK) must always include a name, but the other verbs need not. For example, in the case of (*MARK:A)(*PRUNE) the name inserted is "A", but for (*MARK:A)(*PRUNE:B) the relevant name is "B". This facility can be used to perform simple simultaneous substitutions, as this **pcre2test** example shows:

```
  /(*MARK:pear)apple|(*MARK:orange)lemon/g,replace=${*MARK}
      apple lemon
   2: pear orange
```

PCRE2_SUBSTITUTE_GLOBAL causes the function to iterate over the subject string, replacing every matching substring. If this option is not set, only the first matching substring is replaced. The search for matches takes place in the original subject string (that is, previous replacements do not affect it). Iteration is implemented by advancing the *startoffset* value for each search, which is always passed the entire subject string. If an offset limit is set in the match context, searching stops when that limit is reached.

You can restrict the effect of a global substitution to a portion of the subject string by setting either or both of *startoffset* and an offset limit. Here is a **pcre2test** example:

```
  /B/g,replace=!,use_offset_limit
  ABC ABC ABC ABC\=offset=3,offset_limit=12
   2: ABC A!C A!C ABC
```

When continuing with global substitutions after matching a substring with zero length, an attempt to find a non-empty match at the same offset is performed. If this is not successful, the offset is advanced by one character except when CRLF is a valid newline sequence and the next two characters are CR, LF. In this case, the offset is advanced by two characters.

PCRE2_SUBSTITUTE_UNKNOWN_UNSET causes references to capture groups that do not appear in the pattern to be treated as unset groups. This option should be used with care, because it means that a typo in a group name or number no longer causes the PCRE2_ERROR_NOSUBSTRING error.

PCRE2_SUBSTITUTE_UNSET_EMPTY causes unset capture groups (including unknown groups when PCRE2_SUBSTITUTE_UNKNOWN_UNSET is set) to be treated as empty strings when inserted as described above. If this option is not set, an attempt to insert an unset group causes the PCRE2_ERROR_UNSET error. This option does not influence the extended substitution syntax described below.

PCRE2_SUBSTITUTE_EXTENDED causes extra processing to be applied to the replacement string. Without this option, only the dollar character is special, and only the group insertion forms listed above are valid. When PCRE2_SUBSTITUTE_EXTENDED is set, two things change:

Firstly, backslash in a replacement string is interpreted as an escape character. The usual forms such as \n or \x{ddd} can be used to specify particular character codes, and backslash followed by any non-alphanumeric character quotes that character. Extended quoting can be coded using \Q...\E, exactly as in pattern strings.

There are also four escape sequences for forcing the case of inserted letters. The insertion mechanism has three states: no case forcing, force upper case, and force lower case. The escape sequences change the current state: \U and \L change to upper or lower case forcing, respectively, and \E (when not terminating a \Q quoted sequence) reverts to no case forcing. The sequences \u and \l force the next character (if it is a letter) to upper or lower case, respectively, and then the state automatically reverts to no case forcing. Case forcing applies to all inserted characters, including those from capture groups and letters within \Q...\E quoted sequences. If either PCRE2_UTF or PCRE2_UCP was set when the pattern was compiled, Unicode properties are used for case forcing characters whose code points are greater than 127.

Note that case forcing sequences such as \U...\E do not nest. For example, the result of processing "\Uaa\LBB\Ecc\E" is "AAbbcc"; the final \E has no effect. Note also that the PCRE2_ALT_BSUX and PCRE2_EXTRA_ALT_BSUX options do not apply to replacement strings.

The second effect of setting PCRE2_SUBSTITUTE_EXTENDED is to add more flexibility to capture group substitution. The syntax is similar to that used by Bash:

```
${<n>:-<string>}
${<n>:+<string1>:<string2>}
```

As before, <n> may be a group number or a name. The first form specifies a default value. If group <n> is set, its value is inserted; if not, <string> is expanded and the result inserted. The second form specifies strings that are expanded and inserted when group <n> is set or unset, respectively. The first form is just a convenient shorthand for

```
${<n>:+${<n>}:<string>}
```

Backslash can be used to escape colons and closing curly brackets in the replacement strings. A change of the case forcing state within a replacement string remains in force afterwards, as shown in this **pcre2test** example:

```
/(some)?(body)/substitute_extended,replace=${1:+\U:\L}HeLLo
   body
 1: hello
   somebody
 1: HELLO
```

The PCRE2_SUBSTITUTE_UNSET_EMPTY option does not affect these extended substitutions.

However, PCRE2_SUBSTITUTE_UNKNOWN_UNSET does cause unknown groups in the extended syntax forms to be treated as unset.

If PCRE2_SUBSTITUTE_LITERAL is set, PCRE2_SUBSTITUTE_UNKNOWN_UNSET, PCRE2_SUBSTITUTE_UNSET_EMPTY, and PCRE2_SUBSTITUTE_EXTENDED are irrelevant and are ignored.

Substitution errors

In the event of an error, **pcre2_substitute()** returns a negative error code. Except for PCRE2_ERROR_NOMATCH (which is never returned), errors from **pcre2_match()** are passed straight back.

PCRE2_ERROR_NOSUBSTRING is returned for a non-existent substring insertion, unless PCRE2_SUBSTITUTE_UNKNOWN_UNSET is set.

PCRE2_ERROR_UNSET is returned for an unset substring insertion (including an unknown substring when PCRE2_SUBSTITUTE_UNKNOWN_UNSET is set) when the simple (non-extended) syntax is used and PCRE2_SUBSTITUTE_UNSET_EMPTY is not set.

PCRE2_ERROR_NOMEMORY is returned if the output buffer is not big enough. If the PCRE2_SUBSTITUTE_OVERFLOW_LENGTH option is set, the size of buffer that is needed is returned via *outlengthptr*. Note that this does not happen by default.

PCRE2_ERROR_NULL is returned if PCRE2_SUBSTITUTE_MATCHED is set but the *match_data* argument is NULL.

PCRE2_ERROR_BADREPLACEMENT is used for miscellaneous syntax errors in the replacement string, with more particular errors being PCRE2_ERROR_BADREPESCAPE (invalid escape sequence), PCRE2_ERROR_REPMISSINGBRACE (closing curly bracket not found), PCRE2_ERROR_BADSUBSTITUTION (syntax error in extended group substitution), and PCRE2_ERROR_BADSUBSPATTERN (the pattern match ended before it started or the match started earlier than the current position in the subject, which can happen if \K is used in an assertion).

As for all PCRE2 errors, a text message that describes the error can be obtained by calling the **pcre2_get_error_message()** function (see "Obtaining a textual error message" above).

Substitution callouts

int pcre2_set_substitute_callout(pcre2_match_context ****mcontext*,**
  int (**callout_function***)(pcre2_substitute_callout_block *, void *),**
  void ****callout_data*);**

The **pcre2_set_substitution_callout()** function can be used to specify a callout function for **pcre2_substitute()**. This information is passed in a match context. The callout function is called after each substitution has been processed, but it can cause the replacement not to happen. The callout function is not called for simulated substitutions that happen as a result of the PCRE2_SUBSTITUTE_OVERFLOW_LENGTH option.

The first argument of the callout function is a pointer to a substitute callout block structure, which contains the following fields, not necessarily in this order:

```
  uint32_t    version;
  uint32_t    subscount;
  PCRE2_SPTR  input;
  PCRE2_SPTR  output;
  PCRE2_SIZE *ovector;
  uint32_t    oveccount;
  PCRE2_SIZE  output_offsets[2];
```

The *version* field contains the version number of the block format. The current version is 0. The version number will increase in future if more fields are added, but the intention is never to remove any of the existing fields.

The *subscount* field is the number of the current match. It is 1 for the first callout, 2 for the second, and so on. The *input* and *output* pointers are copies of the values passed to **pcre2_substitute()**.

The *ovector* field points to the ovector, which contains the result of the most recent match. The *oveccount* field contains the number of pairs that are set in the ovector, and is always greater than zero.

The *output_offsets* vector contains the offsets of the replacement in the output string. This has already been processed for dollar and (if requested) backslash substitutions as described above.

The second argument of the callout function is the value passed as *callout_data* when the function was registered. The value returned by the callout function is interpreted as follows:

If the value is zero, the replacement is accepted, and, if PCRE2_SUBSTITUTE_GLOBAL is set, processing continues with a search for the next match. If the value is not zero, the current replacement is not accepted. If the value is greater than zero, processing continues when PCRE2_SUBSTITUTE_GLOBAL is set. Otherwise (the value is less than zero or PCRE2_SUBSTITUTE_GLOBAL is not set), the the rest of the input is copied to the output and the call to **pcre2_substitute()** exits, returning the number of matches so far.

DUPLICATE CAPTURE GROUP NAMES

int pcre2_substring_nametable_scan(const pcre2_code ****code*,**
  PCRE2_SPTR *name***, PCRE2_SPTR ****first***, PCRE2_SPTR ****last***);**

When a pattern is compiled with the PCRE2_DUPNAMES option, names for capture groups are not required to be unique. Duplicate names are always allowed for groups with the same number, created by using the (?| feature. Indeed, if such groups are named, they are required to use the same names.

Normally, patterns that use duplicate names are such that in any one match, only one of each set of identically-named groups participates. An example is shown in the **pcre2pattern** documentation.

When duplicates are present, **pcre2_substring_copy_byname()** and **pcre2_substring_get_byname()** return the first substring corresponding to the given name that is set. Only if none are set is PCRE2_ERROR_UNSET is returned. The **pcre2_substring_number_from_name()** function returns the error PCRE2_ERROR_NOUNIQUESUBSTRING when there are duplicate names.

If you want to get full details of all captured substrings for a given name, you must use the **pcre2_substring_nametable_scan()** function. The first argument is the compiled pattern, and the second is the name. If the third and fourth arguments are NULL, the function returns a group number for a unique name, or PCRE2_ERROR_NOUNIQUESUBSTRING otherwise.

When the third and fourth arguments are not NULL, they must be pointers to variables that 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, and the function returns the length of each entry in code units. In both cases, PCRE2_ERROR_NOSUBSTRING is returned if there are no entries for the given name.

The format of the name table is described above in the section entitled *Information about a pattern*. Given all the relevant entries for the name, you can extract each of their numbers, and hence the captured data.

FINDING ALL POSSIBLE MATCHES AT ONE POSITION

The traditional matching function uses a similar algorithm to Perl, which stops when it finds the first match at a given point in the subject. If you want to find all possible matches, or the longest possible match at a given position, consider using the alternative matching function (see below) instead. If you cannot use the alternative function, you can kludge it up by making use of the callout facility, which is described in the **pcre2callout** 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 **pcre2_match()** to backtrack and try other alternatives. Ultimately, when it runs out of matches, **pcre2_match()** will yield PCRE2_ERROR_NOMATCH.

MATCHING A PATTERN: THE ALTERNATIVE FUNCTION

int pcre2_dfa_match(const pcre2_code ***_code_, PCRE2_SPTR** _subject_**,**
PCRE2_SIZE _length_**, PCRE2_SIZE** _startoffset_**,**
uint32_t _options_**, pcre2_match_data ***_match_data_**,**
pcre2_match_context ***_mcontext_,**
int ***_workspace_, PCRE2_SIZE** _wscount_**);**

The function **pcre2_dfa_match()** is called to match a subject string against a compiled pattern, using a matching algorithm that scans the subject string just once (not counting lookaround assertions), and does not backtrack. This has different characteristics to the normal algorithm, and is not compatible with Perl. Some of the features of PCRE2 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 **pcre2_dfa_match()** does not support, see the **pcre2matching** documentation.

The arguments for the **pcre2_dfa_match()** function are the same as for **pcre2_match()**, plus two extras. The ovector within the match data block is used in a different way, and this is described below. The other common arguments are used in the same way as for **pcre2_match()**, 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 is needed for patterns and subjects where there are a lot of potential matches.

Here is an example of a simple call to **pcre2_dfa_match()**:

```
  int wspace[20];
  pcre2_match_data *md = pcre2_match_data_create(4, NULL);
  int rc = pcre2_dfa_match(
    re,             /* result of pcre2_compile() */
    "some string",  /* the subject string */
    11,             /* the length of the subject string */
    0,              /* start at offset 0 in the subject */
    0,              /* default options */
    md,             /* the match data block */
    NULL,           /* a match context; NULL means use defaults */
    wspace,         /* working space vector */
    20);            /* number of elements (NOT size in bytes) */
```

Option bits for pcre_dfa_match()

The unused bits of the _options_ argument for **pcre2_dfa_match()** must be zero. The only bits that may be set are PCRE2_ANCHORED, PCRE2_COPY_MATCHED_SUBJECT, PCRE2_ENDANCHORED, PCRE2_NOTBOL, PCRE2_NOTEOL, PCRE2_NOTEMPTY, PCRE2_NOTEMPTY_ATSTART, PCRE2_NO_UTF_CHECK, PCRE2_PARTIAL_HARD, PCRE2_PARTIAL_SOFT, PCRE2_DFA_SHORTEST, and PCRE2_DFA_RESTART. All but the last four of these are exactly the same as for **pcre2_match()**, so their description is not repeated here.

```
  PCRE2_PARTIAL_HARD
  PCRE2_PARTIAL_SOFT
```

These have the same general effect as they do for **pcre2_match()**, but the details are slightly different. When PCRE2_PARTIAL_HARD is set for **pcre2_dfa_match()**, it returns PCRE2_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 already been found. When PCRE2_PARTIAL_SOFT is set, the return code PCRE2_ERROR_NOMATCH is converted into PCRE2_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 **pcre2partial** documentation.

```
  PCRE2_DFA_SHORTEST
```

Setting the PCRE2_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.

```
  PCRE2_DFA_RESTART
```

When **pcre2_dfa_match()** returns a partial match, it is possible to call it again, with additional subject characters, and have it continue with the same match. The PCRE2_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 **pcre2partial** documentation.

Successful returns from pcre2_dfa_match()

When **pcre2_dfa_match()** 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 else> <something further>
  <something> <something else>
  <something>
```

On success, the yield of the function is a number greater than zero, which is the number of matched substrings. The offsets of the substrings are returned in the ovector, and can be extracted by number in the same way as for **pcre2_match()**, but the numbers bear no relation to any capture groups that may exist in the pattern, because DFA matching does not support capturing.

Calls to the convenience functions that extract substrings by name return the error PCRE2_ERROR_DFA_UFUNC (unsupported function) if used after a DFA match. The convenience functions that extract substrings by number never return PCRE2_ERROR_NOSUBSTRING.

The matched strings are stored in the ovector in reverse order of length; that is, the longest matching string is first. If there were too many matches to fit into the ovector, the yield of the function is zero, and the vector is filled with the longest matches.

NOTE: PCRE2'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++". 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 such as "a\d+?" or set the PCRE2_NO_AUTO_POSSESS option when compiling.

Error returns from pcre2_dfa_match()

The **pcre2_dfa_match()** function returns a negative number when it fails. Many of the errors are the same as for **pcre2_match()**, as described above. There are in addition the following errors that are specific to **pcre2_dfa_match()**:

```
  PCRE2_ERROR_DFA_UITEM
```

This return is given if **pcre2_dfa_match()** encounters an item in the pattern that it does not support, for instance, the use of \C in a UTF mode or a backreference.

```
  PCRE2_ERROR_DFA_UCOND
```

This return is given if **pcre2_dfa_match()** encounters a condition item that uses a backreference for the condition, or a test for recursion in a specific capture group. These are not supported.

```
  PCRE2_ERROR_DFA_UINVALID_UTF
```

This return is given if **pcre2_dfa_match()** is called for a pattern that was compiled with PCRE2_MATCH_INVALID_UTF. This is not supported for DFA matching.

```
  PCRE2_ERROR_DFA_WSSIZE
```

This return is given if **pcre2_dfa_match()** runs out of space in the *workspace* vector.

```
  PCRE2_ERROR_DFA_RECURSE
```

When a recursion or subroutine call is processed, the matching function calls itself recursively, using private memory for the ovector and *workspace*. This error is given if the internal ovector is not large enough. This should be extremely rare, as a vector of size 1000 is used.

```
  PCRE2_ERROR_DFA_BADRESTART
```

When **pcre2_dfa_match()** is called with the **PCRE2_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

pcre2build(3), **pcre2callout**(3), **pcre2demo(3)**, **pcre2matching**(3), **pcre2partial**(3), **pcre2posix**(3), **pcre2sample**(3), **pcre2unicode**(3).

AUTHOR

Philip Hazel
Retired from University Computing Service
Cambridge, England.

REVISION

Last updated: 30 August 2021