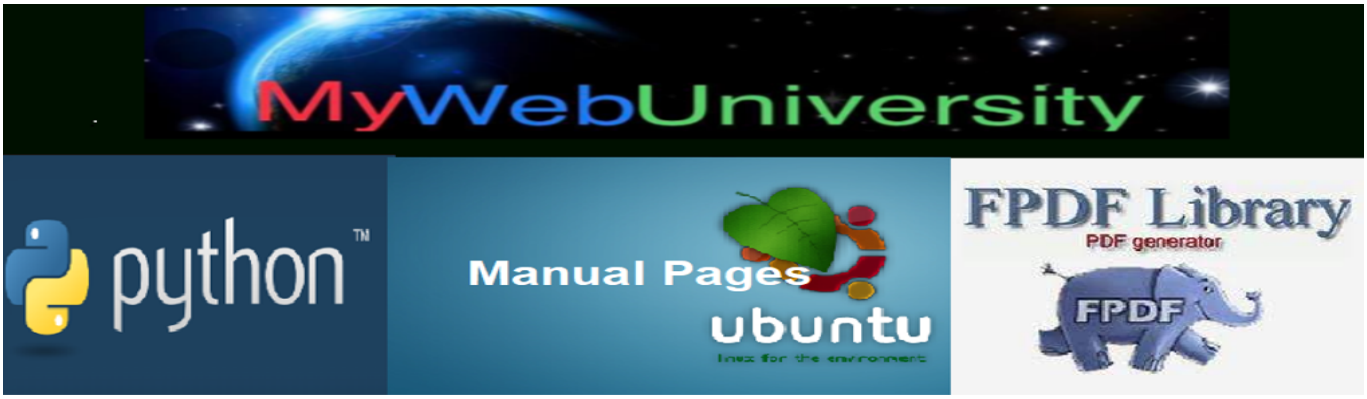

Rocky Enterprise Linux 9.2 Manual Pages on command 'perlapi.1'

$ man perlapi.1

PERLAPI(1) Perl Programmers Reference Guide PERLAPI(1)

NAME

perlapi - autogenerated documentation for the perl public API

DESCRIPTION

This file contains most of the documentation of the perl public API, as generated by embed.pl. Specifically, it is a listing of functions, macros, flags, and variables that may be used by extension writers. Besides perlintern and config.h, some items are listed here as being actually documented in another pod.

At the end is a list of functions which have yet to be documented. Patches welcome! The interfaces of these are subject to change without notice.

Some of the functions documented here are consolidated so that a single entry serves for multiple functions which all do basically the same thing, but have some slight differences. For example, one form might process magic, while another doesn't. The name of each variation is listed at the top of the single entry. But if all have the same signature (arguments and return type) except for their names, only the usage for the base form is shown. If any one of the forms has a different signature (such as returning "const" or not) every function's signature is explicitly displayed.

Anything not listed here or in the other mentioned pods is not part of the public API, and should not be used by extension writers at all. For these reasons, blindly using functions listed in proto.h is to be avoided when writing extensions.

In Perl, unlike C, a string of characters may generally contain embedded "NUL" characters. Sometimes in the documentation a Perl string is referred to as a "buffer" to distinguish it from a C string, but sometimes they are both just referred to as strings.

Note that all Perl API global variables must be referenced with the "PL_" prefix. Again, those not listed here are not to be used by extension writers, and can be changed or removed without notice; same with macros. Some macros are provided for compatibility with the older, unadorned names, but this support may be disabled in a future release.

Perl was originally written to handle US-ASCII only (that is characters whose ordinal numbers are in the range 0 - 127). And documentation and comments may still use the term ASCII, when sometimes in fact the entire range from 0 - 255 is meant.

The non-ASCII characters below 256 can have various meanings, depending on various things. (See, most notably, perllocale.) But usually the whole range can be referred to as ISO-8859-1. Often, the term "Latin-1" (or "Latin1") is used as an equivalent for ISO-8859-1. But some people treat "Latin1" as referring just to the characters in the range 128 through 255, or sometimes from 160 through 255. This documentation uses "Latin1" and "Latin-1" to refer to all 256 characters.

Note that Perl can be compiled and run under either ASCII or EBCDIC (See perlebcdic). Most of the documentation (and even comments in the code) ignore the EBCDIC possibility. For almost all purposes the differences are transparent. As an example, under EBCDIC, instead of UTF-8, UTF-EBCDIC is used to encode Unicode strings, and so whenever this documentation refers to "utf8" (and variants of that name, including in function names), it also (essentially transparently) means "UTF-EBCDIC". But the ordinals of characters differ between ASCII, EBCDIC, and the UTF- encodings, and a string encoded in UTF-EBCDIC may occupy a different number of bytes than in UTF-8.

The organization of this document is tentative and subject to change. Suggestions and patches welcome perl5-porters@perl.org <mailto:perl5-porters@perl.org>.

The sections in this document currently are

"AV Handling"

"Callback Functions"

"Casting"

"Character case changing"

"Character classification"

"Compiler and Preprocessor information"

"Compiler directives"

"Compile-time scope hooks"

"Concurrency"

"COP Hint Hashes"

"Custom Operators"

"CV Handling"

"Debugging"

"Display functions"

"Embedding and Interpreter Cloning"

"Errno"

"Exception Handling (simple) Macros"

"Filesystem configuration values"

"Floating point configuration values"

"Formats"

"General Configuration"

"Global Variables"

"GV Handling"

"Hook manipulation"

"HV Handling"

"Input/Output"

"Integer configuration values"

"Lexer interface"

"Locales"

"Magic"

"Memory Management"

"MRO"

"Multicall Functions"

"Numeric Functions"

"Optree construction"

"Optree Manipulation Functions"

"Pack and Unpack"

"Pad Data Structures"

"Password and Group access"

"Paths to system commands"

"Prototype information"

"REGEXP Functions"

"Signals"

"Site configuration"

"Sockets configuration values"

"Source Filters"

"Stack Manipulation Macros"

"String Handling"

"SV Flags"

"SV Handling"

"Time"

"Typedef names"

"Unicode Support"

"Utility Functions"

"Versioning"

"Warning and Dieing"

"XS"

"Undocumented elements"

The listing below is alphabetical, case insensitive.

AV Handling

"AV"

Described in perlguts.

"AvARRAY"

Returns a pointer to the AV's internal SV* array.

This is useful for doing pointer arithmetic on the array. If all you need is to look up an array element, then prefer "av_fetch".

```
SV**  AvARRAY(AV* av)
```

"av_clear"

Frees all the elements of an array, leaving it empty. The XS equivalent of "@array = ()". See also "av_undef".

Note that it is possible that the actions of a destructor called directly or indirectly by freeing an element of the array could cause the reference count of the array itself to be reduced (e.g. by deleting an entry in the symbol table). So it is a possibility that the AV could have been freed (or even reallocated) on return from the call unless you hold a reference to it.

```
 void  av_clear(AV *av)
```

"av_count"

Returns the number of elements in the array "av". This is the true length of the array, including any undefined elements. It is always the same as "av_top_index(av)?+?1".

```
 Size_t  av_count(AV *av)
```

"av_create_and_push"

NOTE: "av_create_and_push" is experimental and may change or be removed without notice.

Push an SV onto the end of the array, creating the array if necessary. A small internal helper function to remove a commonly duplicated idiom.

NOTE: "av_create_and_push" must be explicitly called as "Perl_av_create_and_push" with an "aTHX_" parameter.

```
 void  Perl_av_create_and_push(pTHX_ AV **const avp,
                               SV *const val)
```

"av_create_and_unshift_one"

NOTE: "av_create_and_unshift_one" is experimental and may change or be removed without notice.

Unshifts an SV onto the beginning of the array, creating the array if necessary. A small internal helper function to remove a commonly duplicated idiom.

NOTE: "av_create_and_unshift_one" must be explicitly called as "Perl_av_create_and_unshift_one" with an "aTHX_" parameter.

```
 SV**  Perl_av_create_and_unshift_one(pTHX_ AV **const avp,
                                     SV *const val)
```

"av_delete"

Deletes the element indexed by "key" from the array, makes the element mortal, and returns it. If "flags" equals "G_DISCARD", the element is freed and NULL is returned. NULL is also returned if "key" is out of range.

Perl equivalent: "splice(@myarray,?$key,?1,?undef)" (with the "splice" in void context if "G_DISCARD" is present).

```
 SV*  av_delete(AV *av, SSize_t key, I32 flags)
```

"av_exists"

Returns true if the element indexed by "key" has been initialized.

This relies on the fact that uninitialized array elements are set to "NULL".

Perl equivalent: "exists($myarray[$key])".

```
bool  av_exists(AV *av, SSize_t key)
```

"av_extend"

Pre-extend an array so that it is capable of storing values at indexes "0..key". Thus "av_extend(av,99)" guarantees that the array can store 100 elements, i.e. that "av_store(av, 0, sv)" through "av_store(av, 99, sv)" on a plain array will work without any further memory allocation.

If the av argument is a tied array then will call the "EXTEND" tied array method with an argument of "(key+1)".

```
void  av_extend(AV *av, SSize_t key)
```

"av_fetch"

Returns the SV at the specified index in the array. The "key" is the index. If lval is true, you are guaranteed to get a real SV back (in case it wasn't real before), which you can then modify. Check that the return value is non-null before dereferencing it to a "SV*".

See "Understanding the Magic of Tied Hashes and Arrays" in perlguts for more information on how to use this function on tied arrays.

The rough perl equivalent is $myarray[$key].

```
SV**  av_fetch(AV *av, SSize_t key, I32 lval)
```

"AvFILL"

Same as "av_top_index" or "av_tindex".

```
SSize_t  AvFILL(AV* av)
```

"av_fill"

Set the highest index in the array to the given number, equivalent to Perl's "$#array?=?$fill;".

The number of elements in the array will be "fill?+?1" after "av_fill()" returns. If the array was previously shorter, then the additional elements appended are set to NULL. If the array was longer, then the excess elements are freed. "av_fill(av,?-1)" is the same as "av_clear(av)".

```
void  av_fill(AV *av, SSize_t fill)
```

"av_len"

Same as "av_top_index". Note that, unlike what the name implies, it returns the

maximum index in the array. This is unlike "sv_len", which returns what you would expect.

To get the true number of elements in the array, instead use "av_count".

```
SSize_t  av_len(AV *av)
```

"av_make"

Creates a new AV and populates it with a list of SVs. The SVs are copied into the array, so they may be freed after the call to "av_make". The new AV will have a reference count of 1.

Perl equivalent: "my @new_array = ($scalar1, $scalar2, $scalar3...);"

```
AV*  av_make(SSize_t size, SV **strp)
```

"av_pop"

Removes one SV from the end of the array, reducing its size by one and returning the SV (transferring control of one reference count) to the caller. Returns &PL_sv_undef if the array is empty.

Perl equivalent: "pop(@myarray);"

```
SV*  av_pop(AV *av)
```

"av_push"

Pushes an SV (transferring control of one reference count) onto the end of the array. The array will grow automatically to accommodate the addition.

Perl equivalent: "push @myarray, $val;".

```
void  av_push(AV *av, SV *val)
```

"av_shift"

Removes one SV from the start of the array, reducing its size by one and returning the SV (transferring control of one reference count) to the caller. Returns &PL_sv_undef if the array is empty.

Perl equivalent: "shift(@myarray);"

```
SV*  av_shift(AV *av)
```

"av_store"

Stores an SV in an array. The array index is specified as "key". The return value will be "NULL" if the operation failed or if the value did not need to be actually stored within the array (as in the case of tied arrays). Otherwise, it can be dereferenced to get the "SV*" that was stored there (= "val")).

Note that the caller is responsible for suitably incrementing the reference count of

"val" before the call, and decrementing it if the function returned "NULL".

Approximate Perl equivalent: "splice(@myarray, $key, 1, $val)".

See "Understanding the Magic of Tied Hashes and Arrays" in perlguts for more information on how to use this function on tied arrays.

```
SV**  av_store(AV *av, SSize_t key, SV *val)
```

"av_tindex"

"av_top_index"

These behave identically. If the array "av" is empty, these return -1; otherwise they return the maximum value of the indices of all the array elements which are currently defined in "av".

They process 'get' magic.

The Perl equivalent for these is $#av.

Use "av_count" to get the number of elements in an array.

```
SSize_t  av_tindex(AV *av)
```

"av_undef"

Undefines the array. The XS equivalent of "undef(@array)".

As well as freeing all the elements of the array (like "av_clear()"), this also frees the memory used by the av to store its list of scalars.

See "av_clear" for a note about the array possibly being invalid on return.

```
void  av_undef(AV *av)
```

"av_unshift"

Unshift the given number of "undef" values onto the beginning of the array. The array will grow automatically to accommodate the addition.

Perl equivalent: "unshift?@myarray,?((undef)?x?$num);"

```
void  av_unshift(AV *av, SSize_t num)
```

"get_av"

Returns the AV of the specified Perl global or package array with the given name (so it won't work on lexical variables). "flags" are passed to "gv_fetchpv". If "GV_ADD" is set and the Perl variable does not exist then it will be created. If "flags" is zero and the variable does not exist then NULL is returned.

Perl equivalent: "@{"$name"}".

NOTE: the "perl_get_av()" form is deprecated.

```
AV*  get_av(const char *name, I32 flags)
```

"newAV"

Creates a new AV. The reference count is set to 1.

Perl equivalent: "my @array;".

```
AV*  newAV()
```

"Nullav"

"DEPRECATED!" It is planned to remove "Nullav" from a future release of Perl. Do not use it for new code; remove it from existing code.

Null AV pointer.

(deprecated - use "(AV *)NULL" instead)

Callback Functions

"call_argv"

Performs a callback to the specified named and package-scoped Perl subroutine with "argv" (a "NULL"-terminated array of strings) as arguments. See perlcall.

Approximate Perl equivalent: "&{"$sub_name"}(@$argv)".

NOTE: the "perl_call_argv()" form is deprecated.

```
I32  call_argv(const char* sub_name, I32 flags, char** argv)
```

"call_method"

Performs a callback to the specified Perl method. The blessed object must be on the stack. See perlcall.

NOTE: the "perl_call_method()" form is deprecated.

```
I32  call_method(const char* methname, I32 flags)
```

"call_pv"

Performs a callback to the specified Perl sub. See perlcall.

NOTE: the "perl_call_pv()" form is deprecated.

```
I32  call_pv(const char* sub_name, I32 flags)
```

"call_sv"

Performs a callback to the Perl sub specified by the SV.

If neither the "G_METHOD" nor "G_METHOD_NAMED" flag is supplied, the SV may be any of a CV, a GV, a reference to a CV, a reference to a GV or "SvPV(sv)" will be used as the name of the sub to call.

If the "G_METHOD" flag is supplied, the SV may be a reference to a CV or "SvPV(sv)" will be used as the name of the method to call.

If the "G_METHOD_NAMED" flag is supplied, "SvPV(sv)" will be used as the name of the

method to call.

Some other values are treated specially for internal use and should not be depended on.

See perlcall.

NOTE: the "perl_call_sv()" form is deprecated.

```
 I32  call_sv(SV* sv, volatile I32 flags)
```

"ENTER"

Opening bracket on a callback. See "LEAVE" and perlcall.

```
    ENTER;
```

"ENTER_with_name"

Same as "ENTER", but when debugging is enabled it also associates the given literal string with the new scope.

```
    ENTER_with_name("name");
```

"eval_pv"

Tells Perl to "eval" the given string in scalar context and return an SV* result.

NOTE: the "perl_eval_pv()" form is deprecated.

```
 SV*  eval_pv(const char* p, I32 croak_on_error)
```

"eval_sv"

Tells Perl to "eval" the string in the SV. It supports the same flags as "call_sv", with the obvious exception of "G_EVAL". See perlcall.

The "G_RETHROW" flag can be used if you only need eval_sv() to execute code specified by a string, but not catch any errors.

NOTE: the "perl_eval_sv()" form is deprecated.

```
 I32  eval_sv(SV* sv, I32 flags)
```

"FREETMPS"

Closing bracket for temporaries on a callback. See "SAVETMPS" and perlcall.

```
    FREETMPS;
```

"G_ARRAY"

Described in perlcall.

"G_DISCARD"

Described in perlcall.

"G_EVAL"

Described in perlcall.

"GIMME"

"DEPRECATED!" It is planned to remove "GIMME" from a future release of Perl. Do not use it for new code; remove it from existing code.

A backward-compatible version of "GIMME_V" which can only return "G_SCALAR" or "G_ARRAY"; in a void context, it returns "G_SCALAR". Deprecated. Use "GIMME_V" instead.

```
U32  GIMME
```

"GIMME_V"

The XSUB-writer's equivalent to Perl's "wantarray". Returns "G_VOID", "G_SCALAR" or "G_ARRAY" for void, scalar or list context, respectively. See perlcall for a usage example.

```
U32  GIMME_V
```

"G_KEEPERR"

Described in perlcall.

"G_NOARGS"

Described in perlcall.

"G_SCALAR"

Described in perlcall.

"G_VOID"

Described in perlcall.

"LEAVE"

Closing bracket on a callback. See "ENTER" and perlcall.

```
LEAVE;
```

"LEAVE_with_name"

Same as "LEAVE", but when debugging is enabled it first checks that the scope has the given name. "name" must be a literal string.

```
LEAVE_with_name("name");
```

"PL_errgv"

Described in perlcall.

"SAVETMPS"

Opening bracket for temporaries on a callback. See "FREETMPS" and perlcall.

```
SAVETMPS;
```

"cBOOL"

Cast-to-bool. A simple "(bool)?expr" cast may not do the right thing: if "bool" is defined as "char", for example, then the cast from "int" is implementation-defined. "(bool)!!(cbool)" in a ternary triggers a bug in xlc on AIX

```
bool  cBOOL(bool expr)
```

"I_32"

Cast an NV to I32 while avoiding undefined C behavior

```
I32  I_32(NV what)
```

"INT2PTR"

Described in perlguts.

```
type  INT2PTR(type, int value)
```

"I_V"

Cast an NV to IV while avoiding undefined C behavior

```
IV  I_V(NV what)
```

"Perl_cpeep_t"

Described in perlguts.

"PTR2IV"

Described in perlguts.

```
IV  PTR2IV(void * ptr)
```

"PTR2nat"

Described in perlguts.

```
IV  PTR2nat(void *)
```

"PTR2NV"

Described in perlguts.

```
NV  PTR2NV(void * ptr)
```

"PTR2ul"

Described in perlguts.

```
unsigned long  PTR2ul(void *)
```

"PTR2UV"

Described in perlguts.

```
UV  PTR2UV(void * ptr)
```

"PTRV"

Described in perlguts.

"U_32"

Cast an NV to U32 while avoiding undefined C behavior

U32 U_32(NV what)

"U_V"

Cast an NV to UV while avoiding undefined C behavior

UV U_V(NV what)

"XOP"

Described in perlguts.

Character case changing

Perl uses "full" Unicode case mappings. This means that converting a single character to another case may result in a sequence of more than one character. For example, the uppercase of "?" (LATIN SMALL LETTER SHARP S) is the two character sequence "SS". This presents some complications The lowercase of all characters in the range 0..255 is a single character, and thus "toLOWER_L1" is furnished. But, "toUPPER_L1" can't exist, as it couldn't return a valid result for all legal inputs. Instead "toUPPER_uvchr" has an API that does allow every possible legal result to be returned.) Likewise no other function that is crippled by not being able to give the correct results for the full range of possible inputs has been implemented here.

"toFOLD"

Converts the specified character to foldcase. If the input is anything but an ASCII uppercase character, that input character itself is returned. Variant "toFOLD_A" is equivalent. (There is no equivalent "to_FOLD_L1" for the full Latin1 range, as the full generality of "toFOLD_uvchr" is needed there.)

U8 toFOLD(U8 ch)

"toFOLD_utf8"

"toFOLD_utf8_safe"

Converts the first UTF-8 encoded character in the sequence starting at "p" and extending no further than "e?-?1" to its foldcase version, and stores that in UTF-8 in "s", and its length in bytes in "lenp". Note that the buffer pointed to by "s" needs to be at least "UTF8_MAXBYTES_CASE+1" bytes since the foldcase version may be longer than the original character.

The first code point of the foldcased version is returned (but note, as explained at the top of this section, that there may be more).

It will not attempt to read beyond "e?-?1", provided that the constraint "s?<?e" is true (this is asserted for in "-DDEBUGGING" builds). If the UTF-8 for the input character is malformed in some way, the program may croak, or the function may return the REPLACEMENT CHARACTER, at the discretion of the implementation, and subject to change in future releases.

"toFOLD_utf8_safe" is now just a different spelling of plain "toFOLD_utf8"

UV toFOLD_utf8(U8* p, U8* e, U8* s, STRLEN* lenp)

"toFOLD_uvchr"

Converts the code point "cp" to its foldcase version, and stores that in UTF-8 in "s", and its length in bytes in "lenp". The code point is interpreted as native if less than 256; otherwise as Unicode. Note that the buffer pointed to by "s" needs to be at least "UTF8_MAXBYTES_CASE+1" bytes since the foldcase version may be longer than the original character.

The first code point of the foldcased version is returned (but note, as explained at the top of this section, that there may be more).

UV toFOLD_uvchr(UV cp, U8* s, STRLEN* lenp)

"toLOWER"

"toLOWER_A"

"toLOWER_L1"

"toLOWER_LATIN1"

"toLOWER_LC"

"toLOWER_uvchr"

"toLOWER_utf8"

"toLOWER_utf8_safe"

These all return the lowercase of a character. The differences are what domain they operate on, and whether the input is specified as a code point (those forms with a "cp" parameter) or as a UTF-8 string (the others). In the latter case, the code point to use is the first one in the buffer of UTF-8 encoded code points, delineated by the arguments "p?..?e?-?1".

"toLOWER" and "toLOWER_A" are synonyms of each other. They return the lowercase of any uppercase ASCII-range code point. All other inputs are returned unchanged. Since these are macros, the input type may be any integral one, and the output will occupy the same number of bits as the input.

"toLOWER_L1" and "toLOWER_LATIN1" are synonyms of each other. They behave identically as "toLOWER" for ASCII-range input. But additionally will return the lowercase of any uppercase code point in the entire 0..255 range, assuming a Latin-1 encoding (or the EBCDIC equivalent on such platforms).

"toLOWER_LC" returns the lowercase of the input code point according to the rules of the current POSIX locale. Input code points outside the range 0..255 are returned unchanged.

"toLOWER_uvchr" returns the lowercase of any Unicode code point. The return value is identical to that of "toLOWER_L1" for input code points in the 0..255 range. The lowercase of the vast majority of Unicode code points is the same as the code point itself. For these, and for code points above the legal Unicode maximum, this returns the input code point unchanged. It additionally stores the UTF-8 of the result into the buffer beginning at "s", and its length in bytes into *lenp. The caller must have made "s" large enough to contain at least "UTF8_MAXBYTES_CASE+1" bytes to avoid possible overflow.

NOTE: the lowercase of a code point may be more than one code point. The return value of this function is only the first of these. The entire lowercase is returned in "s".

To determine if the result is more than a single code point, you can do something like this:

```
 uc = toLOWER_uvchr(cp, s, &len);
 if (len > UTF8SKIP(s)) { is multiple code points }
 else { is a single code point }
```

"toLOWER_utf8" and "toLOWER_utf8_safe" are synonyms of each other. The only difference between these and "toLOWER_uvchr" is that the source for these is encoded in UTF-8, instead of being a code point. It is passed as a buffer starting at "p", with "e" pointing to one byte beyond its end. The "p" buffer may certainly contain more than one code point; but only the first one (up through "e?-?1") is examined. If the UTF-8 for the input character is malformed in some way, the program may croak, or the function may return the REPLACEMENT CHARACTER, at the discretion of the implementation, and subject to change in future releases.

```
 UV  toLOWER          (UV cp)
 UV  toLOWER_A        (UV cp)
 UV  toLOWER_L1       (UV cp)
```

UV toLOWER_LATIN1   (UV cp)

UV toLOWER_LC   (UV cp)

UV toLOWER_uvchr   (UV cp, U8* s, STRLEN* lenp)

UV toLOWER_utf8   (U8* p, U8* e, U8* s, STRLEN* lenp)

UV toLOWER_utf8_safe(U8* p, U8* e, U8* s, STRLEN* lenp)

"toTITLE"

Converts the specified character to titlecase. If the input is anything but an ASCII lowercase character, that input character itself is returned. Variant "toTITLE_A" is equivalent. (There is no "toTITLE_L1" for the full Latin1 range, as the full generality of "toTITLE_uvchr" is needed there. Titlecase is not a concept used in locale handling, so there is no functionality for that.)

U8 toTITLE(U8 ch)

"toTITLE_utf8"

"toTITLE_utf8_safe"

Convert the first UTF-8 encoded character in the sequence starting at "p" and extending no further than "e?-?1" to its titlecase version, and stores that in UTF-8 in "s", and its length in bytes in "lenp". Note that the buffer pointed to by "s" needs to be at least "UTF8_MAXBYTES_CASE+1" bytes since the titlecase version may be longer than the original character.

The first code point of the titlecased version is returned (but note, as explained at the top of this section, that there may be more).

It will not attempt to read beyond "e?-?1", provided that the constraint "s?<?e" is true (this is asserted for in "-DDEBUGGING" builds). If the UTF-8 for the input character is malformed in some way, the program may croak, or the function may return the REPLACEMENT CHARACTER, at the discretion of the implementation, and subject to change in future releases.

"toTITLE_utf8_safe" is now just a different spelling of plain "toTITLE_utf8"

UV toTITLE_utf8(U8* p, U8* e, U8* s, STRLEN* lenp)

"toTITLE_uvchr"

Converts the code point "cp" to its titlecase version, and stores that in UTF-8 in "s", and its length in bytes in "lenp". The code point is interpreted as native if less than 256; otherwise as Unicode. Note that the buffer pointed to by "s" needs to be at least "UTF8_MAXBYTES_CASE+1" bytes since the titlecase version may be longer

than the original character.

The first code point of the titlecased version is returned (but note, as explained at the top of this section, that there may be more).

UV toTITLE_uvchr(UV cp, U8* s, STRLEN* lenp)

"toUPPER"

Converts the specified character to uppercase. If the input is anything but an ASCII lowercase character, that input character itself is returned. Variant "toUPPER_A" is equivalent.

U8 toUPPER(int ch)

"toUPPER_utf8"

"toUPPER_utf8_safe"

Converts the first UTF-8 encoded character in the sequence starting at "p" and extending no further than "e?-?1" to its uppercase version, and stores that in UTF-8 in "s", and its length in bytes in "lenp". Note that the buffer pointed to by "s" needs to be at least "UTF8_MAXBYTES_CASE+1" bytes since the uppercase version may be longer than the original character.

The first code point of the uppercased version is returned (but note, as explained at the top of this section, that there may be more).

It will not attempt to read beyond "e?-?1", provided that the constraint "s?<?e" is true (this is asserted for in "-DDEBUGGING" builds). If the UTF-8 for the input character is malformed in some way, the program may croak, or the function may return the REPLACEMENT CHARACTER, at the discretion of the implementation, and subject to change in future releases.

"toUPPER_utf8_safe" is now just a different spelling of plain "toUPPER_utf8"

UV toUPPER_utf8(U8* p, U8* e, U8* s, STRLEN* lenp)

"toUPPER_uvchr"

Converts the code point "cp" to its uppercase version, and stores that in UTF-8 in "s", and its length in bytes in "lenp". The code point is interpreted as native if less than 256; otherwise as Unicode. Note that the buffer pointed to by "s" needs to be at least "UTF8_MAXBYTES_CASE+1" bytes since the uppercase version may be longer than the original character.

The first code point of the uppercased version is returned (but note, as explained at the top of this section, that there may be more.)

UV toUPPER_uvchr(UV cp, U8* s, STRLEN* lenp)

Character classification

This section is about functions (really macros) that classify characters into types, such as punctuation versus alphabetic, etc. Most of these are analogous to regular expression character classes. (See "POSIX Character Classes" in perlrecharclass.) There are several variants for each class. (Not all macros have all variants; each item below lists the ones valid for it.) None are affected by "use bytes", and only the ones with "LC" in the name are affected by the current locale.

The base function, e.g., "isALPHA()", takes any signed or unsigned value, treating it as a code point, and returns a boolean as to whether or not the character represented by it is (or on non-ASCII platforms, corresponds to) an ASCII character in the named class based on platform, Unicode, and Perl rules. If the input is a number that doesn't fit in an octet, FALSE is returned.

Variant "isFOO_A" (e.g., "isALPHA_A()") is identical to the base function with no suffix "_A". This variant is used to emphasize by its name that only ASCII-range characters can return TRUE.

Variant "isFOO_L1" imposes the Latin-1 (or EBCDIC equivalent) character set onto the platform. That is, the code points that are ASCII are unaffected, since ASCII is a subset of Latin-1. But the non-ASCII code points are treated as if they are Latin-1 characters. For example, "isWORDCHAR_L1()" will return true when called with the code point 0xDF, which is a word character in both ASCII and EBCDIC (though it represents different characters in each). If the input is a number that doesn't fit in an octet, FALSE is returned. (Perl's documentation uses a colloquial definition of Latin-1, to include all code points below 256.)

Variant "isFOO_uvchr" is exactly like the "isFOO_L1" variant, for inputs below 256, but if the code point is larger than 255, Unicode rules are used to determine if it is in the character class. For example, "isWORDCHAR_uvchr(0x100)" returns TRUE, since 0x100 is LATIN CAPITAL LETTER A WITH MACRON in Unicode, and is a word character.

Variants "isFOO_utf8" and "isFOO_utf8_safe" are like "isFOO_uvchr", but are used for UTF-8 encoded strings. The two forms are different names for the same thing. Each call to one of these classifies the first character of the string starting at "p". The second parameter, "e", points to anywhere in the string beyond the first character, up to one byte past the end of the entire string. Although both variants are identical, the suffix

"_safe" in one name emphasizes that it will not attempt to read beyond "e?-?1", provided that the constraint "s?<?e" is true (this is asserted for in "-DDEBUGGING" builds). If the UTF-8 for the input character is malformed in some way, the program may croak, or the function may return FALSE, at the discretion of the implementation, and subject to change in future releases.

Variant "isFOO_LC" is like the "isFOO_A" and "isFOO_L1" variants, but the result is based on the current locale, which is what "LC" in the name stands for. If Perl can determine that the current locale is a UTF-8 locale, it uses the published Unicode rules; otherwise, it uses the C library function that gives the named classification. For example, "isDIGIT_LC()" when not in a UTF-8 locale returns the result of calling "isdigit()". FALSE is always returned if the input won't fit into an octet. On some platforms where the C library function is known to be defective, Perl changes its result to follow the POSIX standard's rules.

Variant "isFOO_LC_uvchr" acts exactly like "isFOO_LC" for inputs less than 256, but for larger ones it returns the Unicode classification of the code point.

Variants "isFOO_LC_utf8" and "isFOO_LC_utf8_safe" are like "isFOO_LC_uvchr", but are used for UTF-8 encoded strings. The two forms are different names for the same thing. Each call to one of these classifies the first character of the string starting at "p". The second parameter, "e", points to anywhere in the string beyond the first character, up to one byte past the end of the entire string. Although both variants are identical, the suffix "_safe" in one name emphasizes that it will not attempt to read beyond "e?-?1", provided that the constraint "s?<?e" is true (this is asserted for in "-DDEBUGGING" builds). If the UTF-8 for the input character is malformed in some way, the program may croak, or the function may return FALSE, at the discretion of the implementation, and subject to change in future releases.

"isALPHA"

"isALPHA_A"

"isALPHA_L1"

"isALPHA_uvchr"

"isALPHA_utf8_safe"

"isALPHA_utf8"

"isALPHA_LC"

"isALPHA_LC_uvchr"

"isALPHA_LC_utf8_safe"

Returns a boolean indicating whether the specified input is one of "[A-Za-z]", analogous to "m/[[:alpha:]]/". See the top of this section for an explanation of the variants.

```
bool  isALPHA             (UV ch)
bool  isALPHA_A           (UV ch)
bool  isALPHA_L1          (UV ch)
bool  isALPHA_uvchr       (UV ch)
bool  isALPHA_utf8_safe   (U8 * s, U8 * end)
bool  isALPHA_utf8        (U8 * s, U8 * end)
bool  isALPHA_LC          (UV ch)
bool  isALPHA_LC_uvchr    (UV ch)
bool  isALPHA_LC_utf8_safe(U8 * s, U8 *end)
```

"isALPHANUMERIC"

"isALPHANUMERIC_A"

"isALPHANUMERIC_L1"

"isALPHANUMERIC_uvchr"

"isALPHANUMERIC_utf8_safe"

"isALPHANUMERIC_utf8"

"isALPHANUMERIC_LC"

"isALPHANUMERIC_LC_uvchr"

"isALPHANUMERIC_LC_utf8_safe"

"isALNUMC"

"isALNUMC_A"

"isALNUMC_L1"

"isALNUMC_LC"

"isALNUMC_LC_uvchr"

Returns a boolean indicating whether the specified character is one of "[A-Za-z0-9]", analogous to "m/[[:alnum:]]/". See the top of this section for an explanation of the variants.

A (discouraged from use) synonym is "isALNUMC" (where the "C" suffix means this corresponds to the C language alphanumeric definition). Also there are the variants "isALNUMC_A", "isALNUMC_L1" "isALNUMC_LC", and "isALNUMC_LC_uvchr".

```
    bool  isALPHANUMERIC            (UV ch)
    bool  isALPHANUMERIC_A          (UV ch)
    bool  isALPHANUMERIC_L1         (UV ch)
    bool  isALPHANUMERIC_uvchr      (UV ch)
    bool  isALPHANUMERIC_utf8_safe  (U8 * s, U8 * end)
    bool  isALPHANUMERIC_utf8       (U8 * s, U8 * end)
    bool  isALPHANUMERIC_LC         (UV ch)
    bool  isALPHANUMERIC_LC_uvchr   (UV ch)
    bool  isALPHANUMERIC_LC_utf8_safe(U8 * s, U8 *end)
    bool  isALNUMC                  (UV ch)
    bool  isALNUMC_A                (UV ch)
    bool  isALNUMC_L1               (UV ch)
    bool  isALNUMC_LC               (UV ch)
    bool  isALNUMC_LC_uvchr         (UV ch)
"isASCII"
"isASCII_A"
"isASCII_L1"
"isASCII_uvchr"
"isASCII_utf8_safe"
"isASCII_utf8"
"isASCII_LC"
"isASCII_LC_uvchr"
"isASCII_LC_utf8_safe"
   Returns a boolean indicating whether the specified character is one of the 128
   characters in the ASCII character set, analogous to "m/[[:ascii:]]/".  On non-ASCII
   platforms, it returns TRUE iff this character corresponds to an ASCII character.
   Variants "isASCII_A()" and "isASCII_L1()" are identical to "isASCII()".  See the top
   of this section for an explanation of the variants.  Note, however, that some
   platforms do not have the C library routine "isascii()".  In these cases, the variants
   whose names contain "LC" are the same as the corresponding ones without.
   Also note, that because all ASCII characters are UTF-8 invariant (meaning they have
   the exact same representation (always a single byte) whether encoded in UTF-8 or not),
   "isASCII" will give the correct results when called with any byte in any string
```

encoded or not in UTF-8. And similarly "isASCII_utf8" and "isASCII_utf8_safe" will work properly on any string encoded or not in UTF-8.

```
bool  isASCII             (UV ch)
bool  isASCII_A           (UV ch)
bool  isASCII_L1          (UV ch)
bool  isASCII_uvchr       (UV ch)
bool  isASCII_utf8_safe   (U8 * s, U8 * end)
bool  isASCII_utf8        (U8 * s, U8 * end)
bool  isASCII_LC          (UV ch)
bool  isASCII_LC_uvchr    (UV ch)
bool  isASCII_LC_utf8_safe(U8 * s, U8 *end)
```

"isBLANK"

"isBLANK_A"

"isBLANK_L1"

"isBLANK_uvchr"

"isBLANK_utf8_safe"

"isBLANK_utf8"

"isBLANK_LC"

"isBLANK_LC_uvchr"

"isBLANK_LC_utf8_safe"

Returns a boolean indicating whether the specified character is a character considered to be a blank, analogous to "m/[[:blank:]]/". See the top of this section for an explanation of the variants. Note, however, that some platforms do not have the C library routine "isblank()". In these cases, the variants whose names contain "LC" are the same as the corresponding ones without.

```
bool  isBLANK             (UV ch)
bool  isBLANK_A           (UV ch)
bool  isBLANK_L1          (UV ch)
bool  isBLANK_uvchr       (UV ch)
bool  isBLANK_utf8_safe   (U8 * s, U8 * end)
bool  isBLANK_utf8        (U8 * s, U8 * end)
bool  isBLANK_LC          (UV ch)
bool  isBLANK_LC_uvchr    (UV ch)
```

```
    bool  isBLANK_LC_utf8_safe(U8 * s, U8 *end)
```

"isCNTRL"

"isCNTRL_A"

"isCNTRL_L1"

"isCNTRL_uvchr"

"isCNTRL_utf8_safe"

"isCNTRL_utf8"

"isCNTRL_LC"

"isCNTRL_LC_uvchr"

"isCNTRL_LC_utf8_safe"

Returns a boolean indicating whether the specified character is a control character, analogous to "m/[[:cntrl:]]/". See the top of this section for an explanation of the variants. On EBCDIC platforms, you almost always want to use the "isCNTRL_L1" variant.

```
    bool  isCNTRL             (UV ch)
    bool  isCNTRL_A           (UV ch)
    bool  isCNTRL_L1          (UV ch)
    bool  isCNTRL_uvchr       (UV ch)
    bool  isCNTRL_utf8_safe   (U8 * s, U8 * end)
    bool  isCNTRL_utf8        (U8 * s, U8 * end)
    bool  isCNTRL_LC          (UV ch)
    bool  isCNTRL_LC_uvchr    (UV ch)
    bool  isCNTRL_LC_utf8_safe(U8 * s, U8 *end)
```

"isDIGIT"

"isDIGIT_A"

"isDIGIT_L1"

"isDIGIT_uvchr"

"isDIGIT_utf8_safe"

"isDIGIT_utf8"

"isDIGIT_LC"

"isDIGIT_LC_uvchr"

"isDIGIT_LC_utf8_safe"

Returns a boolean indicating whether the specified character is a digit, analogous to

"m/[[:digit:]]/". Variants "isDIGIT_A" and "isDIGIT_L1" are identical to "isDIGIT".

See the top of this section for an explanation of the variants.

```
bool  isDIGIT             (UV ch)
bool  isDIGIT_A           (UV ch)
bool  isDIGIT_L1          (UV ch)
bool  isDIGIT_uvchr       (UV ch)
bool  isDIGIT_utf8_safe   (U8 * s, U8 * end)
bool  isDIGIT_utf8        (U8 * s, U8 * end)
bool  isDIGIT_LC          (UV ch)
bool  isDIGIT_LC_uvchr    (UV ch)
bool  isDIGIT_LC_utf8_safe(U8 * s, U8 *end)
```

"isGRAPH"

"isGRAPH_A"

"isGRAPH_L1"

"isGRAPH_uvchr"

"isGRAPH_utf8_safe"

"isGRAPH_utf8"

"isGRAPH_LC"

"isGRAPH_LC_uvchr"

"isGRAPH_LC_utf8_safe"

Returns a boolean indicating whether the specified character is a graphic character, analogous to "m/[[:graph:]]/". See the top of this section for an explanation of the variants.

```
bool  isGRAPH             (UV ch)
bool  isGRAPH_A           (UV ch)
bool  isGRAPH_L1          (UV ch)
bool  isGRAPH_uvchr       (UV ch)
bool  isGRAPH_utf8_safe   (U8 * s, U8 * end)
bool  isGRAPH_utf8        (U8 * s, U8 * end)
bool  isGRAPH_LC          (UV ch)
bool  isGRAPH_LC_uvchr    (UV ch)
bool  isGRAPH_LC_utf8_safe(U8 * s, U8 *end)
```

"isIDCONT"

"isIDCONT_A"

"isIDCONT_L1"

"isIDCONT_uvchr"

"isIDCONT_utf8_safe"

"isIDCONT_utf8"

"isIDCONT_LC"

"isIDCONT_LC_uvchr"

"isIDCONT_LC_utf8_safe"

Returns a boolean indicating whether the specified character can be the second or succeeding character of an identifier. This is very close to, but not quite the same as the official Unicode property "XID_Continue". The difference is that this returns true only if the input character also matches "isWORDCHAR". See the top of this section for an explanation of the variants.

```
bool  isIDCONT            (UV ch)
bool  isIDCONT_A          (UV ch)
bool  isIDCONT_L1         (UV ch)
bool  isIDCONT_uvchr      (UV ch)
bool  isIDCONT_utf8_safe  (U8 * s, U8 * end)
bool  isIDCONT_utf8       (U8 * s, U8 * end)
bool  isIDCONT_LC         (UV ch)
bool  isIDCONT_LC_uvchr   (UV ch)
bool  isIDCONT_LC_utf8_safe(U8 * s, U8 *end)
```

"isIDFIRST"

"isIDFIRST_A"

"isIDFIRST_L1"

"isIDFIRST_uvchr"

"isIDFIRST_utf8_safe"

"isIDFIRST_utf8"

"isIDFIRST_LC"

"isIDFIRST_LC_uvchr"

"isIDFIRST_LC_utf8_safe"

Returns a boolean indicating whether the specified character can be the first character of an identifier. This is very close to, but not quite the same as the

official Unicode property "XID_Start". The difference is that this returns true only if the input character also matches "isWORDCHAR". See the top of this section for an explanation of the variants.

```
 bool  isIDFIRST             (UV ch)
 bool  isIDFIRST_A           (UV ch)
 bool  isIDFIRST_L1          (UV ch)
 bool  isIDFIRST_uvchr       (UV ch)
 bool  isIDFIRST_utf8_safe   (U8 * s, U8 * end)
 bool  isIDFIRST_utf8        (U8 * s, U8 * end)
 bool  isIDFIRST_LC          (UV ch)
 bool  isIDFIRST_LC_uvchr    (UV ch)
 bool  isIDFIRST_LC_utf8_safe(U8 * s, U8 *end)
```

"isLOWER"

"isLOWER_A"

"isLOWER_L1"

"isLOWER_uvchr"

"isLOWER_utf8_safe"

"isLOWER_utf8"

"isLOWER_LC"

"isLOWER_LC_uvchr"

"isLOWER_LC_utf8_safe"

Returns a boolean indicating whether the specified character is a lowercase character, analogous to "m/[[:lower:]]/". See the top of this section for an explanation of the variants

```
 bool  isLOWER             (UV ch)
 bool  isLOWER_A           (UV ch)
 bool  isLOWER_L1          (UV ch)
 bool  isLOWER_uvchr       (UV ch)
 bool  isLOWER_utf8_safe   (U8 * s, U8 * end)
 bool  isLOWER_utf8        (U8 * s, U8 * end)
 bool  isLOWER_LC          (UV ch)
 bool  isLOWER_LC_uvchr    (UV ch)
 bool  isLOWER_LC_utf8_safe(U8 * s, U8 *end)
```

"isOCTAL"

"isOCTAL_A"

"isOCTAL_L1"

Returns a boolean indicating whether the specified character is an octal digit, [0-7].

The only two variants are "isOCTAL_A" and "isOCTAL_L1"; each is identical to "isOCTAL".

```
bool  isOCTAL(UV ch)
```

"isPRINT"

"isPRINT_A"

"isPRINT_L1"

"isPRINT_uvchr"

"isPRINT_utf8_safe"

"isPRINT_utf8"

"isPRINT_LC"

"isPRINT_LC_uvchr"

"isPRINT_LC_utf8_safe"

Returns a boolean indicating whether the specified character is a printable character, analogous to "m/[[:print:]]/". See the top of this section for an explanation of the variants.

```
bool  isPRINT             (UV ch)
bool  isPRINT_A           (UV ch)
bool  isPRINT_L1          (UV ch)
bool  isPRINT_uvchr       (UV ch)
bool  isPRINT_utf8_safe   (U8 * s, U8 * end)
bool  isPRINT_utf8        (U8 * s, U8 * end)
bool  isPRINT_LC          (UV ch)
bool  isPRINT_LC_uvchr    (UV ch)
bool  isPRINT_LC_utf8_safe(U8 * s, U8 *end)
```

"isPSXSPC"

"isPSXSPC_A"

"isPSXSPC_L1"

"isPSXSPC_uvchr"

"isPSXSPC_utf8_safe"

"isPSXSPC_utf8"

"isPSXSPC_LC"

"isPSXSPC_LC_uvchr"

"isPSXSPC_LC_utf8_safe"

(short for Posix Space) Starting in 5.18, this is identical in all its forms to the corresponding "isSPACE()" macros. The locale forms of this macro are identical to their corresponding "isSPACE()" forms in all Perl releases. In releases prior to 5.18, the non-locale forms differ from their "isSPACE()" forms only in that the "isSPACE()" forms don't match a Vertical Tab, and the "isPSXSPC()" forms do. Otherwise they are identical. Thus this macro is analogous to what "m/[[:space:]]/" matches in a regular expression. See the top of this section for an explanation of the variants.

```
bool  isPSXSPC            (UV ch)
bool  isPSXSPC_A          (UV ch)
bool  isPSXSPC_L1         (UV ch)
bool  isPSXSPC_uvchr      (UV ch)
bool  isPSXSPC_utf8_safe  (U8 * s, U8 * end)
bool  isPSXSPC_utf8       (U8 * s, U8 * end)
bool  isPSXSPC_LC         (UV ch)
bool  isPSXSPC_LC_uvchr   (UV ch)
bool  isPSXSPC_LC_utf8_safe(U8 * s, U8 *end)
```

"isPUNCT"

"isPUNCT_A"

"isPUNCT_L1"

"isPUNCT_uvchr"

"isPUNCT_utf8_safe"

"isPUNCT_utf8"

"isPUNCT_LC"

"isPUNCT_LC_uvchr"

"isPUNCT_LC_utf8_safe"

Returns a boolean indicating whether the specified character is a punctuation character, analogous to "m/[[:punct:]]/". Note that the definition of what is punctuation isn't as straightforward as one might desire. See "POSIX Character

Classes" in perlrecharclass for details. See the top of this section for an explanation of the variants.

```
bool  isPUNCT             (UV ch)
bool  isPUNCT_A           (UV ch)
bool  isPUNCT_L1          (UV ch)
bool  isPUNCT_uvchr       (UV ch)
bool  isPUNCT_utf8_safe   (U8 * s, U8 * end)
bool  isPUNCT_utf8        (U8 * s, U8 * end)
bool  isPUNCT_LC          (UV ch)
bool  isPUNCT_LC_uvchr    (UV ch)
bool  isPUNCT_LC_utf8_safe(U8 * s, U8 *end)
```

"isSPACE"

"isSPACE_A"

"isSPACE_L1"

"isSPACE_uvchr"

"isSPACE_utf8_safe"

"isSPACE_utf8"

"isSPACE_LC"

"isSPACE_LC_uvchr"

"isSPACE_LC_utf8_safe"

Returns a boolean indicating whether the specified character is a whitespace character. This is analogous to what "m/\s/" matches in a regular expression. Starting in Perl 5.18 this also matches what "m/[[:space:]]/" does. Prior to 5.18, only the locale forms of this macro (the ones with "LC" in their names) matched precisely what "m/[[:space:]]/" does. In those releases, the only difference, in the non-locale variants, was that "isSPACE()" did not match a vertical tab. (See "isPSXSPC" for a macro that matches a vertical tab in all releases.) See the top of this section for an explanation of the variants.

```
bool  isSPACE             (UV ch)
bool  isSPACE_A           (UV ch)
bool  isSPACE_L1          (UV ch)
bool  isSPACE_uvchr       (UV ch)
bool  isSPACE_utf8_safe   (U8 * s, U8 * end)
```

```
    bool  isSPACE_utf8       (U8 * s, U8 * end)
    bool  isSPACE_LC         (UV ch)
    bool  isSPACE_LC_uvchr    (UV ch)
    bool  isSPACE_LC_utf8_safe(U8 * s, U8 *end)
"isUPPER"
"isUPPER_A"
"isUPPER_L1"
"isUPPER_uvchr"
"isUPPER_utf8_safe"
"isUPPER_utf8"
"isUPPER_LC"
"isUPPER_LC_uvchr"
"isUPPER_LC_utf8_safe"
   Returns a boolean indicating whether the specified character is an uppercase
   character, analogous to "m/[[:upper:]]/".  See the top of this section for an
   explanation of the variants.
    bool  isUPPER            (UV ch)
    bool  isUPPER_A          (UV ch)
    bool  isUPPER_L1         (UV ch)
    bool  isUPPER_uvchr       (UV ch)
    bool  isUPPER_utf8_safe   (U8 * s, U8 * end)
    bool  isUPPER_utf8       (U8 * s, U8 * end)
    bool  isUPPER_LC         (UV ch)
    bool  isUPPER_LC_uvchr    (UV ch)
    bool  isUPPER_LC_utf8_safe(U8 * s, U8 *end)
"isWORDCHAR"
"isWORDCHAR_A"
"isWORDCHAR_L1"
"isWORDCHAR_uvchr"
"isWORDCHAR_utf8_safe"
"isWORDCHAR_utf8"
"isWORDCHAR_LC"
"isWORDCHAR_LC_uvchr"
```

"isWORDCHAR_LC_utf8_safe"

"isALNUM"

"isALNUM_A"

"isALNUM_LC"

"isALNUM_LC_uvchr"

Returns a boolean indicating whether the specified character is a character that is a word character, analogous to what "m/\w/" and "m/[[:word:]]/" match in a regular expression. A word character is an alphabetic character, a decimal digit, a connecting punctuation character (such as an underscore), or a "mark" character that attaches to one of those (like some sort of accent). "isALNUM()" is a synonym provided for backward compatibility, even though a word character includes more than the standard C language meaning of alphanumeric. See the top of this section for an explanation of the variants. "isWORDCHAR_A", "isWORDCHAR_L1", "isWORDCHAR_uvchr", "isWORDCHAR_LC", "isWORDCHAR_LC_uvchr", "isWORDCHAR_LC_utf8", and "isWORDCHAR_LC_utf8_safe" are also as described there, but additionally include the platform's native underscore.

```
 bool  isWORDCHAR             (UV ch)
 bool  isWORDCHAR_A           (UV ch)
 bool  isWORDCHAR_L1          (UV ch)
 bool  isWORDCHAR_uvchr       (UV ch)
 bool  isWORDCHAR_utf8_safe   (U8 * s, U8 * end)
 bool  isWORDCHAR_utf8        (U8 * s, U8 * end)
 bool  isWORDCHAR_LC          (UV ch)
 bool  isWORDCHAR_LC_uvchr    (UV ch)
 bool  isWORDCHAR_LC_utf8_safe(U8 * s, U8 *end)
 bool  isALNUM                (UV ch)
 bool  isALNUM_A              (UV ch)
 bool  isALNUM_LC             (UV ch)
 bool  isALNUM_LC_uvchr       (UV ch)
```

"isXDIGIT"

"isXDIGIT_A"

"isXDIGIT_L1"

"isXDIGIT_uvchr"

"isXDIGIT_utf8_safe"

"isXDIGIT_utf8"

"isXDIGIT_LC"

"isXDIGIT_LC_uvchr"

"isXDIGIT_LC_utf8_safe"

Returns a boolean indicating whether the specified character is a hexadecimal digit. In the ASCII range these are "[0-9A-Fa-f]". Variants "isXDIGIT_A()" and "isXDIGIT_L1()" are identical to "isXDIGIT()". See the top of this section for an explanation of the variants.

```
bool  isXDIGIT             (UV ch)
bool  isXDIGIT_A           (UV ch)
bool  isXDIGIT_L1          (UV ch)
bool  isXDIGIT_uvchr       (UV ch)
bool  isXDIGIT_utf8_safe   (U8 * s, U8 * end)
bool  isXDIGIT_utf8        (U8 * s, U8 * end)
bool  isXDIGIT_LC          (UV ch)
bool  isXDIGIT_LC_uvchr    (UV ch)
bool  isXDIGIT_LC_utf8_safe(U8 * s, U8 *end)
```

Compiler and Preprocessor information

"CPPLAST"

This symbol is intended to be used along with "CPPRUN" in the same manner symbol "CPPMINUS" is used with "CPPSTDIN". It contains either "-" or "".

"CPPMINUS"

This symbol contains the second part of the string which will invoke the C preprocessor on the standard input and produce to standard output. This symbol will have the value "-" if "CPPSTDIN" needs a minus to specify standard input, otherwise the value is "".

"CPPRUN"

This symbol contains the string which will invoke a C preprocessor on the standard input and produce to standard output. It needs to end with "CPPLAST", after all other preprocessor flags have been specified. The main difference with "CPPSTDIN" is that this program will never be a pointer to a shell wrapper, i.e. it will be empty if no preprocessor is available directly to the user. Note that it may well be different

from the preprocessor used to compile the C program.

"CPPSTDIN"

This symbol contains the first part of the string which will invoke the C preprocessor on the standard input and produce to standard output. Typical value of "cc -E" or "/lib/cpp", but it can also call a wrapper. See "CPPRUN".

"HASATTRIBUTE_ALWAYS_INLINE"

Can we handle "GCC" attribute for functions that should always be inlined.

"HASATTRIBUTE_DEPRECATED"

Can we handle "GCC" attribute for marking deprecated "APIs"

"HASATTRIBUTE_FORMAT"

Can we handle "GCC" attribute for checking printf-style formats

"HASATTRIBUTE_NONNULL"

Can we handle "GCC" attribute for nonnull function parms.

"HASATTRIBUTE_NORETURN"

Can we handle "GCC" attribute for functions that do not return

"HASATTRIBUTE_PURE"

Can we handle "GCC" attribute for pure functions

"HASATTRIBUTE_UNUSED"

Can we handle "GCC" attribute for unused variables and arguments

"HASATTRIBUTE_WARN_UNUSED_RESULT"

Can we handle "GCC" attribute for warning on unused results

"HAS_BUILTIN_ADD_OVERFLOW"

This symbol, if defined, indicates that the compiler supports "__builtin_add_overflow" for adding integers with overflow checks.

"HAS_BUILTIN_CHOOSE_EXPR"

Can we handle "GCC" builtin for compile-time ternary-like expressions

"HAS_BUILTIN_EXPECT"

Can we handle "GCC" builtin for telling that certain values are more likely

"HAS_BUILTIN_MUL_OVERFLOW"

This symbol, if defined, indicates that the compiler supports "__builtin_mul_overflow" for multiplying integers with overflow checks.

"HAS_BUILTIN_SUB_OVERFLOW"

This symbol, if defined, indicates that the compiler supports "__builtin_sub_overflow"

for subtracting integers with overflow checks.

"HAS_C99_VARIADIC_MACROS"

If defined, the compiler supports C99 variadic macros.

"HAS_STATIC_INLINE"

This symbol, if defined, indicates that the C compiler supports C99-style static inline. That is, the function can't be called from another translation unit.

"MEM_ALIGNBYTES"

This symbol contains the number of bytes required to align a double, or a long double when applicable. Usual values are 2, 4 and 8. The default is eight, for safety. For cross-compiling or multiarch support, Configure will set a minimum of 8.

"PERL_STATIC_INLINE"

This symbol gives the best-guess incantation to use for static inline functions. If "HAS_STATIC_INLINE" is defined, this will give C99-style inline. If "HAS_STATIC_INLINE" is not defined, this will give a plain 'static'. It will always be defined to something that gives static linkage. Possibilities include

```
static inline       (c99)
static __inline__   (gcc -ansi)
static __inline     (MSVC)
static _inline      (older MSVC)
static              (c89 compilers)
```

"U32_ALIGNMENT_REQUIRED"

This symbol, if defined, indicates that you must access character data through U32-aligned pointers.

Compiler directives

"ASSUME"

"ASSUME" is like "assert()", but it has a benefit in a release build. It is a hint to a compiler about a statement of fact in a function call free expression, which allows the compiler to generate better machine code. In a debug build, ASSUME(x) is a synonym for assert(x). ASSUME(0) means the control path is unreachable. In a for loop, "ASSUME" can be used to hint that a loop will run at least X times. "ASSUME" is based off MSVC's "__assume" intrinsic function, see its documents for more details.

```
ASSUME(bool expr)
```

"dNOOP"

Declare nothing; typically used as a placeholder to replace something that used to declare something. Works on compilers that require declarations before any code.

```
dNOOP;
```

"END_EXTERN_C"

When not compiling using C++, expands to nothing. Otherwise ends a section of code already begun by a "START_EXTERN_C".

```
END_EXTERN_C
```

"EXTERN_C"

When not compiling using C++, expands to nothing. Otherwise is used in a declaration of a function to indicate the function should have external C linkage. This is required for things to work for just about all functions with external linkage compiled into perl. Often, you can use "START_EXTERN_C" ... "END_EXTERN_C" blocks surrounding all your code that you need to have this linkage.

Example usage:

```
EXTERN_C int flock(int fd, int op);
```

"LIKELY"

Returns the input unchanged, but at the same time it gives a branch prediction hint to the compiler that this condition is likely to be true.

```
LIKELY(bool expr)
```

"NOOP"

Do nothing; typically used as a placeholder to replace something that used to do something.

```
NOOP;
```

"PERL_UNUSED_ARG"

This is used to suppress compiler warnings that a parameter to a function is not used. This situation can arise, for example, when a parameter is needed under some configuration conditions, but not others, so that C preprocessor conditional compilation causes it be used just some times.

```
PERL_UNUSED_ARG(void x);
```

"PERL_UNUSED_CONTEXT"

This is used to suppress compiler warnings that the thread context parameter to a function is not used. This situation can arise, for example, when a C preprocessor conditional compilation causes it be used just some times.

PERL_UNUSED_CONTEXT;

"PERL_UNUSED_DECL"

Tells the compiler that the parameter in the function prototype just before it is not necessarily expected to be used in the function. Not that many compilers understand this, so this should only be used in cases where "PERL_UNUSED_ARG" can't conveniently be used.

Example usage:

```
Signal_t
Perl_perly_sighandler(int sig, Siginfo_t *sip PERL_UNUSED_DECL,
                      void *uap PERL_UNUSED_DECL, bool safe)
```

"PERL_UNUSED_RESULT"

This macro indicates to discard the return value of the function call inside it, e.g.,

```
PERL_UNUSED_RESULT(foo(a, b))
```

The main reason for this is that the combination of "gcc -Wunused-result" (part of "-Wall") and the "__attribute__((warn_unused_result))" cannot be silenced with casting to "void". This causes trouble when the system header files use the attribute.

Use "PERL_UNUSED_RESULT" sparingly, though, since usually the warning is there for a good reason: you might lose success/failure information, or leak resources, or changes in resources.

But sometimes you just want to ignore the return value, e.g., on codepaths soon ending up in abort, or in "best effort" attempts, or in situations where there is no good way to handle failures.

Sometimes "PERL_UNUSED_RESULT" might not be the most natural way: another possibility is that you can capture the return value and use "PERL_UNUSED_VAR" on that.

```
PERL_UNUSED_RESULT(void x)
```

"PERL_UNUSED_VAR"

This is used to suppress compiler warnings that the variable x is not used. This situation can arise, for example, when a C preprocessor conditional compilation causes it be used just some times.

```
PERL_UNUSED_VAR(void x);
```

"PERL_USE_GCC_BRACE_GROUPS"

This C pre-processor value, if defined, indicates that it is permissible to use the GCC brace groups extension. This extension, of the form

({ statement ... })

turns the block consisting of statements ... into an expression with a value, unlike plain C language blocks. This can present optimization possibilities, BUT you generally need to specify an alternative in case this ability doesn't exist or has otherwise been forbidden.

Example usage:

```
#ifdef PERL_USE_GCC_BRACE_GROUPS
  ...
#else
  ...
#endif
```

"START_EXTERN_C"

When not compiling using C++, expands to nothing. Otherwise begins a section of code in which every function will effectively have "EXTERN_C" applied to it, that is to have external C linkage. The section is ended by a "END_EXTERN_C".

```
START_EXTERN_C
```

"STATIC"

Described in perlguts.

"STMT_START"

"STMT_END"

This allows a series of statements in a macro to be used as a single statement, as in

```
if (x) STMT_START { ... } STMT_END else ...
```

Note that you can't return a value out of them, which limits their utility. But see "PERL_USE_GCC_BRACE_GROUPS".

"UNLIKELY"

Returns the input unchanged, but at the same time it gives a branch prediction hint to the compiler that this condition is likely to be false.

```
UNLIKELY(bool expr)
```

"__ASSERT_"

This is a helper macro to avoid preprocessor issues, replaced by nothing unless under DEBUGGING, where it expands to an assert of its argument, followed by a comma (hence the comma operator). If we just used a straight assert(), we would get a comma with nothing before it when not DEBUGGING.

__ASSERT_(bool expr)

Compile-time scope hooks

"BhkDISABLE"

NOTE: "BhkDISABLE" is experimental and may change or be removed without notice.

Temporarily disable an entry in this BHK structure, by clearing the appropriate flag.

"which" is a preprocessor token indicating which entry to disable.

void BhkDISABLE(BHK *hk, which)

"BhkENABLE"

NOTE: "BhkENABLE" is experimental and may change or be removed without notice.

Re-enable an entry in this BHK structure, by setting the appropriate flag. "which" is a preprocessor token indicating which entry to enable. This will assert (under -DDEBUGGING) if the entry doesn't contain a valid pointer.

void BhkENABLE(BHK *hk, which)

"BhkENTRY_set"

NOTE: "BhkENTRY_set" is experimental and may change or be removed without notice.

Set an entry in the BHK structure, and set the flags to indicate it is valid. "which" is a preprocessing token indicating which entry to set. The type of "ptr" depends on the entry.

void BhkENTRY_set(BHK *hk, which, void *ptr)

"blockhook_register"

NOTE: "blockhook_register" is experimental and may change or be removed without notice.

Register a set of hooks to be called when the Perl lexical scope changes at compile time. See "Compile-time scope hooks" in perlguts.

NOTE: "blockhook_register" must be explicitly called as "Perl_blockhook_register" with an "aTHX_" parameter.

void Perl_blockhook_register(pTHX_ BHK *hk)

Concurrency

"aTHX"

Described in perlguts.

"aTHX_"

Described in perlguts.

"CPERLscope"

"DEPRECATED!" It is planned to remove "CPERLscope" from a future release of Perl. Do not use it for new code; remove it from existing code.

Now a no-op.

```
void  CPERLscope(void x)
```

"dTHR"

Described in perlguts.

"dTHX"

Described in perlguts.

"dTHXa"

On threaded perls, set "pTHX" to "a"; on unthreaded perls, do nothing

"dTHXoa"

Now a synonym for "dTHXa".

"dVAR"

This is now a synonym for dNOOP: declare nothing

"GETENV_PRESERVES_OTHER_THREAD"

This symbol, if defined, indicates that the getenv system call doesn't zap the static buffer of "getenv()" in a different thread. The typical "getenv()" implementation will return a pointer to the proper position in **environ. But some may instead copy them to a static buffer in "getenv()". If there is a per-thread instance of that buffer, or the return points to **environ, then a many-reader/1-writer mutex will work; otherwise an exclusive locking mutex is required to prevent races.

"HAS_PTHREAD_ATFORK"

This symbol, if defined, indicates that the "pthread_atfork" routine is available to setup fork handlers.

"HAS_PTHREAD_ATTR_SETSCOPE"

This symbol, if defined, indicates that the "pthread_attr_setscope" system call is available to set the contention scope attribute of a thread attribute object.

"HAS_PTHREAD_YIELD"

This symbol, if defined, indicates that the "pthread_yield" routine is available to yield the execution of the current thread. "sched_yield" is preferable to "pthread_yield".

"HAS_SCHED_YIELD"

This symbol, if defined, indicates that the "sched_yield" routine is available to

yield the execution of the current thread. "sched_yield" is preferable to

"pthread_yield".

"I_MACH_CTHREADS"

This symbol, if defined, indicates to the C program that it should include

mach/cthreads.h.

```
#ifdef I_MACH_CTHREADS
    #include <mach_cthreads.h>
#endif
```

"I_PTHREAD"

This symbol, if defined, indicates to the C program that it should include pthread.h.

```
#ifdef I_PTHREAD
    #include <pthread.h>
#endif
```

"MULTIPLICITY"

This symbol, if defined, indicates that Perl should be built to use multiplicity.

"OLD_PTHREADS_API"

This symbol, if defined, indicates that Perl should be built to use the old draft

"POSIX" threads "API".

"OLD_PTHREAD_CREATE_JOINABLE"

This symbol, if defined, indicates how to create pthread in joinable (aka undetached)

state. "NOTE": not defined if pthread.h already has defined "PTHREAD_CREATE_JOINABLE"

(the new version of the constant). If defined, known values are

"PTHREAD_CREATE_UNDETACHED" and "__UNDETACHED".

"pTHX"

Described in perlguts.

"pTHX_"

Described in perlguts.

"SCHED_YIELD"

This symbol defines the way to yield the execution of the current thread. Known ways

are "sched_yield", "pthread_yield", and "pthread_yield" with "NULL".

"SVf"

Described in perlguts.

"SVfARG"

Described in perlguts.

SVfARG(SV *sv)

COP Hint Hashes

"cop_fetch_label"

NOTE: "cop_fetch_label" is experimental and may change or be removed without notice.

Returns the label attached to a cop, and stores its length in bytes into *len. Upon return, *flags will be set to either "SVf_UTF8" or 0.

Alternatively, use the macro "CopLABEL_len_flags"; or if you don't need to know if the label is UTF-8 or not, the macro "CopLABEL_len"; or if you additionally dont need to know the length, "CopLABEL".

```
const char *  cop_fetch_label(COP *const cop, STRLEN *len,
                              U32 *flags)
```

"CopFILE"

Returns the name of the file associated with the "COP" "c"

```
const char *  CopFILE(const COP * c)
```

"CopFILEAV"

Returns the AV associated with the "COP" "c"

```
AV *  CopFILEAV(const COP * c)
```

"CopFILEGV"

Returns the GV associated with the "COP" "c"

```
GV *  CopFILEGV(const COP * c)
```

"CopFILEGV_set"

Available only on unthreaded perls. Makes "pv" the name of the file associated with the "COP" "c"

```
void  CopFILEGV_set(COP * c, GV * gv)
```

"CopFILE_set"

Makes "pv" the name of the file associated with the "COP" "c"

```
void  CopFILE_set(COP * c, const char * pv)
```

"CopFILESV"

Returns the SV associated with the "COP" "c"

```
SV *  CopFILESV(const COP * c)
```

"cophh_2hv"

NOTE: "cophh_2hv" is experimental and may change or be removed without notice.

Generates and returns a standard Perl hash representing the full set of key/value pairs in the cop hints hash "cophh". "flags" is currently unused and must be zero.

```
HV *  cophh_2hv(const COPHH *cophh, U32 flags)
```

"cophh_copy"

NOTE: "cophh_copy" is experimental and may change or be removed without notice.

Make and return a complete copy of the cop hints hash "cophh".

```
COPHH *  cophh_copy(COPHH *cophh)
```

"cophh_delete_pv"

NOTE: "cophh_delete_pv" is experimental and may change or be removed without notice.

Like "cophh_delete_pvn", but takes a nul-terminated string instead of a string/length pair.

```
COPHH *  cophh_delete_pv(COPHH *cophh, char *key, U32 hash,
                         U32 flags)
```

"cophh_delete_pvn"

NOTE: "cophh_delete_pvn" is experimental and may change or be removed without notice.

Delete a key and its associated value from the cop hints hash "cophh", and returns the modified hash. The returned hash pointer is in general not the same as the hash pointer that was passed in. The input hash is consumed by the function, and the pointer to it must not be subsequently used. Use "cophh_copy" if you need both hashes.

The key is specified by "keypv" and "keylen". If "flags" has the "COPHH_KEY_UTF8" bit set, the key octets are interpreted as UTF-8, otherwise they are interpreted as Latin-1. "hash" is a precomputed hash of the key string, or zero if it has not been precomputed.

```
COPHH *  cophh_delete_pvn(COPHH *cophh, const char *keypv,
                          STRLEN keylen, U32 hash, U32 flags)
```

"cophh_delete_pvs"

NOTE: "cophh_delete_pvs" is experimental and may change or be removed without notice.

Like "cophh_delete_pvn", but takes a literal string instead of a string/length pair, and no precomputed hash.

```
COPHH *  cophh_delete_pvs(COPHH *cophh, "key", U32 flags)
```

"cophh_delete_sv"

NOTE: "cophh_delete_sv" is experimental and may change or be removed without notice.

Like "cophh_delete_pvn", but takes a Perl scalar instead of a string/length pair.

```
 COPHH *  cophh_delete_sv(COPHH *cophh, SV *key, U32 hash,
                          U32 flags)
```

"cophh_exists_pv"

NOTE: "cophh_exists_pv" is experimental and may change or be removed without notice.

Like "cophh_exists_pvn", but takes a nul-terminated string instead of a string/length pair.

```
 bool  cophh_exists_pv(const COPHH *cophh, const char *key,
                       U32 hash, U32 flags)
```

"cophh_exists_pvn"

NOTE: "cophh_exists_pvn" is experimental and may change or be removed without notice.

Look up the entry in the cop hints hash "cophh" with the key specified by "keypv" and "keylen". If "flags" has the "COPHH_KEY_UTF8" bit set, the key octets are interpreted as UTF-8, otherwise they are interpreted as Latin-1. "hash" is a precomputed hash of the key string, or zero if it has not been precomputed. Returns true if a value exists, and false otherwise.

```
 bool  cophh_exists_pvn(const COPHH *cophh, const char *keypv,
                        STRLEN keylen, U32 hash, U32 flags)
```

"cophh_exists_pvs"

NOTE: "cophh_exists_pvs" is experimental and may change or be removed without notice.

Like "cophh_exists_pvn", but takes a literal string instead of a string/length pair, and no precomputed hash.

```
 bool  cophh_exists_pvs(const COPHH *cophh, "key", U32 flags)
```

"cophh_exists_sv"

NOTE: "cophh_exists_sv" is experimental and may change or be removed without notice.

Like "cophh_exists_pvn", but takes a Perl scalar instead of a string/length pair.

```
 bool  cophh_exists_sv(const COPHH *cophh, SV *key, U32 hash,
                       U32 flags)
```

"cophh_fetch_pv"

NOTE: "cophh_fetch_pv" is experimental and may change or be removed without notice.

Like "cophh_fetch_pvn", but takes a nul-terminated string instead of a string/length pair.

```
 SV *  cophh_fetch_pv(const COPHH *cophh, const char *key,
```

U32 hash, U32 flags)

"cophh_fetch_pvn"

NOTE: "cophh_fetch_pvn" is experimental and may change or be removed without notice.

Look up the entry in the cop hints hash "cophh" with the key specified by "keypv" and "keylen". If "flags" has the "COPHH_KEY_UTF8" bit set, the key octets are interpreted as UTF-8, otherwise they are interpreted as Latin-1. "hash" is a precomputed hash of the key string, or zero if it has not been precomputed. Returns a mortal scalar copy of the value associated with the key, or &PL_sv_placeholder if there is no value associated with the key.

```
SV *  cophh_fetch_pvn(const COPHH *cophh, const char *keypv,
                      STRLEN keylen, U32 hash, U32 flags)
```

"cophh_fetch_pvs"

NOTE: "cophh_fetch_pvs" is experimental and may change or be removed without notice.

Like "cophh_fetch_pvn", but takes a literal string instead of a string/length pair, and no precomputed hash.

```
SV *  cophh_fetch_pvs(const COPHH *cophh, "key", U32 flags)
```

"cophh_fetch_sv"

NOTE: "cophh_fetch_sv" is experimental and may change or be removed without notice.

Like "cophh_fetch_pvn", but takes a Perl scalar instead of a string/length pair.

```
SV *  cophh_fetch_sv(const COPHH *cophh, SV *key, U32 hash,
                     U32 flags)
```

"cophh_free"

NOTE: "cophh_free" is experimental and may change or be removed without notice.

Discard the cop hints hash "cophh", freeing all resources associated with it.

```
void  cophh_free(COPHH *cophh)
```

"cophh_new_empty"

NOTE: "cophh_new_empty" is experimental and may change or be removed without notice.

Generate and return a fresh cop hints hash containing no entries.

```
COPHH *  cophh_new_empty()
```

"cophh_store_pv"

NOTE: "cophh_store_pv" is experimental and may change or be removed without notice.

Like "cophh_store_pvn", but takes a nul-terminated string instead of a string/length pair.

```
 COPHH *  cophh_store_pv(COPHH *cophh, const char *key, U32 hash,
                         SV *value, U32 flags)
```

"cophh_store_pvn"

NOTE: "cophh_store_pvn" is experimental and may change or be removed without notice.

Stores a value, associated with a key, in the cop hints hash "cophh", and returns the modified hash. The returned hash pointer is in general not the same as the hash pointer that was passed in. The input hash is consumed by the function, and the pointer to it must not be subsequently used. Use "cophh_copy" if you need both hashes.

The key is specified by "keypv" and "keylen". If "flags" has the "COPHH_KEY_UTF8" bit set, the key octets are interpreted as UTF-8, otherwise they are interpreted as Latin-1. "hash" is a precomputed hash of the key string, or zero if it has not been precomputed.

"value" is the scalar value to store for this key. "value" is copied by this function, which thus does not take ownership of any reference to it, and later changes to the scalar will not be reflected in the value visible in the cop hints hash. Complex types of scalar will not be stored with referential integrity, but will be coerced to strings.

```
 COPHH *  cophh_store_pvn(COPHH *cophh, const char *keypv,
                          STRLEN keylen, U32 hash, SV *value,
                          U32 flags)
```

"cophh_store_pvs"

NOTE: "cophh_store_pvs" is experimental and may change or be removed without notice.

Like "cophh_store_pvn", but takes a literal string instead of a string/length pair, and no precomputed hash.

```
 COPHH *  cophh_store_pvs(COPHH *cophh, "key", SV *value,
                          U32 flags)
```

"cophh_store_sv"

NOTE: "cophh_store_sv" is experimental and may change or be removed without notice.

Like "cophh_store_pvn", but takes a Perl scalar instead of a string/length pair.

```
 COPHH *  cophh_store_sv(COPHH *cophh, SV *key, U32 hash,
                         SV *value, U32 flags)
```

"cop_hints_2hv"

Generates and returns a standard Perl hash representing the full set of hint entries in the cop "cop". "flags" is currently unused and must be zero.

```
HV *  cop_hints_2hv(const COP *cop, U32 flags)
```

"cop_hints_exists_pv"

Like "cop_hints_exists_pvn", but takes a nul-terminated string instead of a string/length pair.

```
bool  cop_hints_exists_pv(const COP *cop, const char *key,
                          U32 hash, U32 flags)
```

"cop_hints_exists_pvn"

Look up the hint entry in the cop "cop" with the key specified by "keypv" and "keylen". If "flags" has the "COPHH_KEY_UTF8" bit set, the key octets are interpreted as UTF-8, otherwise they are interpreted as Latin-1. "hash" is a precomputed hash of the key string, or zero if it has not been precomputed. Returns true if a value exists, and false otherwise.

```
bool  cop_hints_exists_pvn(const COP *cop, const char *keypv,
                           STRLEN keylen, U32 hash, U32 flags)
```

"cop_hints_exists_pvs"

Like "cop_hints_exists_pvn", but takes a literal string instead of a string/length pair, and no precomputed hash.

```
bool  cop_hints_exists_pvs(const COP *cop, "key", U32 flags)
```

"cop_hints_exists_sv"

Like "cop_hints_exists_pvn", but takes a Perl scalar instead of a string/length pair.

```
bool  cop_hints_exists_sv(const COP *cop, SV *key, U32 hash,
                          U32 flags)
```

"cop_hints_fetch_pv"

Like "cop_hints_fetch_pvn", but takes a nul-terminated string instead of a string/length pair.

```
SV *  cop_hints_fetch_pv(const COP *cop, const char *key,
                         U32 hash, U32 flags)
```

"cop_hints_fetch_pvn"

Look up the hint entry in the cop "cop" with the key specified by "keypv" and "keylen". If "flags" has the "COPHH_KEY_UTF8" bit set, the key octets are interpreted as UTF-8, otherwise they are interpreted as Latin-1. "hash" is a precomputed hash of

the key string, or zero if it has not been precomputed. Returns a mortal scalar copy of the value associated with the key, or &PL_sv_placeholder if there is no value associated with the key.

```
SV *  cop_hints_fetch_pvn(const COP *cop, const char *keypv,
                          STRLEN keylen, U32 hash, U32 flags)
```

"cop_hints_fetch_pvs"

Like "cop_hints_fetch_pvn", but takes a literal string instead of a string/length pair, and no precomputed hash.

```
SV *  cop_hints_fetch_pvs(const COP *cop, "key", U32 flags)
```

"cop_hints_fetch_sv"

Like "cop_hints_fetch_pvn", but takes a Perl scalar instead of a string/length pair.

```
SV *  cop_hints_fetch_sv(const COP *cop, SV *key, U32 hash,
                         U32 flags)
```

"CopLABEL"

Returns the label attached to a cop.

```
const char *  CopLABEL(COP *const cop)
```

"CopLABEL_len"

Returns the label attached to a cop, and stores its length in bytes into *len.

```
const char *  CopLABEL_len(COP *const cop, STRLEN *len)
```

"CopLABEL_len_flags"

Returns the label attached to a cop, and stores its length in bytes into *len. Upon return, *flags will be set to either "SVf_UTF8" or 0.

```
const char *  CopLABEL_len_flags(COP *const cop, STRLEN *len,
                                 U32 *flags)
```

"CopLINE"

Returns the line number in the source code associated with the "COP" "c"

```
STRLEN  CopLINE(const COP * c)
```

"CopSTASH"

Returns the stash associated with "c".

```
HV *  CopSTASH(const COP * c)
```

"CopSTASH_eq"

Returns a boolean as to whether or not "hv" is the stash associated with "c".

```
bool  CopSTASH_eq(const COP * c, const HV * hv)
```

"CopSTASHPV"

Returns the package name of the stash associated with "c", or "NULL" if no associated stash

```
char *  CopSTASHPV(const COP * c)
```

"CopSTASHPV_set"

Set the package name of the stash associated with "c", to the NUL-terminated C string "p", creating the package if necessary.

```
void  CopSTASHPV_set(COP * c, const char * pv)
```

"CopSTASH_set"

Set the stash associated with "c" to "hv".

```
bool  CopSTASH_set(COP * c, HV * hv)
```

"cop_store_label"

NOTE: "cop_store_label" is experimental and may change or be removed without notice.

Save a label into a "cop_hints_hash". You need to set flags to "SVf_UTF8" for a UTF-8 label. Any other flag is ignored.

```
void  cop_store_label(COP *const cop, const char *label,
                      STRLEN len, U32 flags)
```

"PERL_SI"

Use this typedef to declare variables that are to hold "struct stackinfo".

Custom Operators

"custom_op_desc"

"DEPRECATED!" It is planned to remove "custom_op_desc" from a future release of Perl. Do not use it for new code; remove it from existing code.

Return the description of a given custom op. This was once used by the "OP_DESC" macro, but is no longer: it has only been kept for compatibility, and should not be used.

```
const char *  custom_op_desc(const OP *o)
```

"custom_op_name"

"DEPRECATED!" It is planned to remove "custom_op_name" from a future release of Perl. Do not use it for new code; remove it from existing code.

Return the name for a given custom op. This was once used by the "OP_NAME" macro, but is no longer: it has only been kept for compatibility, and should not be used.

```
const char *  custom_op_name(const OP *o)
```

"custom_op_register"

Register a custom op. See "Custom Operators" in perlguts.

NOTE: "custom_op_register" must be explicitly called as "Perl_custom_op_register" with an "aTHX_" parameter.

```
void  Perl_custom_op_register(pTHX_ Perl_ppaddr_t ppaddr,
                              const XOP *xop)
```

"Perl_custom_op_xop"

Return the XOP structure for a given custom op. This macro should be considered internal to "OP_NAME" and the other access macros: use them instead. This macro does call a function. Prior to 5.19.6, this was implemented as a function.

```
const XOP *  Perl_custom_op_xop(pTHX_ const OP *o)
```

"XopDISABLE"

Temporarily disable a member of the XOP, by clearing the appropriate flag.

```
void  XopDISABLE(XOP *xop, which)
```

"XopENABLE"

Reenable a member of the XOP which has been disabled.

```
void  XopENABLE(XOP *xop, which)
```

"XopENTRY"

Return a member of the XOP structure. "which" is a cpp token indicating which entry to return. If the member is not set this will return a default value. The return type depends on "which". This macro evaluates its arguments more than once. If you are using "Perl_custom_op_xop" to retrieve a "XOP *" from a "OP *", use the more efficient "XopENTRYCUSTOM" instead.

```
XopENTRY(XOP *xop, which)
```

"XopENTRYCUSTOM"

Exactly like "XopENTRY(XopENTRY(Perl_custom_op_xop(aTHX_ o), which)" but more efficient. The "which" parameter is identical to "XopENTRY".

```
XopENTRYCUSTOM(const OP *o, which)
```

"XopENTRY_set"

Set a member of the XOP structure. "which" is a cpp token indicating which entry to set. See "Custom Operators" in perlguts for details about the available members and how they are used. This macro evaluates its argument more than once.

```
void  XopENTRY_set(XOP *xop, which, value)
```

"XopFLAGS"

Return the XOP's flags.

```
U32  XopFLAGS(XOP *xop)
```

CV Handling

This section documents functions to manipulate CVs which are code-values, meaning subroutines. For more information, see perlguts.

"caller_cx"

The XSUB-writer's equivalent of caller(). The returned "PERL_CONTEXT" structure can be interrogated to find all the information returned to Perl by "caller". Note that XSUBs don't get a stack frame, so "caller_cx(0, NULL)" will return information for the immediately-surrounding Perl code.

This function skips over the automatic calls to &DB::sub made on the behalf of the debugger. If the stack frame requested was a sub called by "DB::sub", the return value will be the frame for the call to "DB::sub", since that has the correct line number/etc. for the call site. If dbcxp is non-"NULL", it will be set to a pointer to the frame for the sub call itself.

```
const PERL_CONTEXT *  caller_cx(I32 level,
                                const PERL_CONTEXT **dbcxp)
```

"CvGV"

Returns the GV associated with the CV "sv", reifying it if necessary.

```
GV *  CvGV(CV *sv)
```

"CvSTASH"

Returns the stash of the CV. A stash is the symbol table hash, containing the package-scoped variables in the package where the subroutine was defined. For more information, see perlguts.

This also has a special use with XS AUTOLOAD subs. See "Autoloading with XSUBs" in perlguts.

```
HV*  CvSTASH(CV* cv)
```

"find_runcv"

Locate the CV corresponding to the currently executing sub or eval. If "db_seqp" is non_null, skip CVs that are in the DB package and populate *db_seqp with the cop sequence number at the point that the DB:: code was entered. (This allows debuggers to eval in the scope of the breakpoint rather than in the scope of the debugger

itself.)

CV* find_runcv(U32 *db_seqp)

"get_cv"

"get_cvs"

"get_cvn_flags"

These return the CV of the specified Perl subroutine. "flags" are passed to "gv_fetchpvn_flags". If "GV_ADD" is set and the Perl subroutine does not exist then it will be declared (which has the same effect as saying "sub name;"). If "GV_ADD" is not set and the subroutine does not exist, then NULL is returned.

The forms differ only in how the subroutine is specified.. With "get_cvs", the name is a literal C string, enclosed in double quotes. With "get_cv", the name is given by the "name" parameter, which must be a NUL-terminated C string. With "get_cvn_flags", the name is also given by the "name" parameter, but it is a Perl string (possibly containing embedded NUL bytes), and its length in bytes is contained in the "len" parameter.

NOTE: the "perl_get_cv()" form is deprecated.

NOTE: the "perl_get_cvs()" form is deprecated.

NOTE: the "perl_get_cvn_flags()" form is deprecated.

CV* get_cv (const char* name, I32 flags)

CV * get_cvs ("string", I32 flags)

CV* get_cvn_flags(const char* name, STRLEN len, I32 flags)

"Nullcv"

"DEPRECATED!" It is planned to remove "Nullcv" from a future release of Perl. Do not use it for new code; remove it from existing code.

Null CV pointer.

(deprecated - use "(CV *)NULL" instead)

Debugging

"dump_all"

Dumps the entire optree of the current program starting at "PL_main_root" to "STDERR". Also dumps the optrees for all visible subroutines in "PL_defstash".

void dump_all()

"dump_c_backtrace"

Dumps the C backtrace to the given "fp".

Returns true if a backtrace could be retrieved, false if not.

```
bool  dump_c_backtrace(PerlIO* fp, int max_depth, int skip)
```

"dump_packsubs"

Dumps the optrees for all visible subroutines in "stash".

```
void  dump_packsubs(const HV* stash)
```

"get_c_backtrace_dump"

Returns a SV containing a dump of "depth" frames of the call stack, skipping the "skip" innermost ones. "depth" of 20 is usually enough.

The appended output looks like:

```
...
1   10e004812:0082   Perl_croak   util.c:1716    /usr/bin/perl
2   10df8d6d2:1d72   perl_parse   perl.c:3975    /usr/bin/perl
...
```

The fields are tab-separated. The first column is the depth (zero being the innermost non-skipped frame). In the hex:offset, the hex is where the program counter was in "S_parse_body", and the :offset (might be missing) tells how much inside the "S_parse_body" the program counter was.

The "util.c:1716" is the source code file and line number.

The /usr/bin/perl is obvious (hopefully).

Unknowns are "-". Unknowns can happen unfortunately quite easily: if the platform doesn't support retrieving the information; if the binary is missing the debug information; if the optimizer has transformed the code by for example inlining.

```
SV*  get_c_backtrace_dump(int max_depth, int skip)
```

"HAS_BACKTRACE"

This symbol, if defined, indicates that the "backtrace()" routine is available to get a stack trace. The execinfo.h header must be included to use this routine.

"op_class"

Given an op, determine what type of struct it has been allocated as. Returns one of the OPclass enums, such as OPclass_LISTOP.

```
OPclass  op_class(const OP *o)
```

"op_dump"

Dumps the optree starting at OP "o" to "STDERR".

```
void  op_dump(const OP *o)
```

"sv_dump"

Dumps the contents of an SV to the "STDERR" filehandle.

For an example of its output, see Devel::Peek.

```
 void  sv_dump(SV* sv)
```

Display functions

"form"

"form_nocontext"

These take a sprintf-style format pattern and conventional (non-SV) arguments and return the formatted string.

```
    (char *) Perl_form(pTHX_ const char* pat, ...)
```

can be used any place a string (char *) is required:

```
    char * s = Perl_form("%d.%d",major,minor);
```

They use a single (per-thread) private buffer so if you want to format several strings you must explicitly copy the earlier strings away (and free the copies when you are done).

The two forms differ only in that "form_nocontext" does not take a thread context ("aTHX") parameter, so is used in situations where the caller doesn't already have the thread context.

NOTE: "form" must be explicitly called as "Perl_form" with an "aTHX_" parameter.

```
 char*  Perl_form     (pTHX_ const char* pat, ...)
 char*  form_nocontext(const char* pat, ...)
```

"mess"

"mess_nocontext"

These take a sprintf-style format pattern and argument list, which are used to generate a string message. If the message does not end with a newline, then it will be extended with some indication of the current location in the code, as described for "mess_sv".

Normally, the resulting message is returned in a new mortal SV. But during global destruction a single SV may be shared between uses of this function.

The two forms differ only in that "mess_nocontext" does not take a thread context ("aTHX") parameter, so is used in situations where the caller doesn't already have the thread context.

NOTE: "mess" must be explicitly called as "Perl_mess" with an "aTHX_" parameter.

```
 SV*  Perl_mess     (pTHX_ const char* pat, ...)
 SV*  mess_nocontext(const char* pat, ...)
```

"mess_sv"

Expands a message, intended for the user, to include an indication of the current location in the code, if the message does not already appear to be complete.

"basemsg" is the initial message or object. If it is a reference, it will be used as-is and will be the result of this function. Otherwise it is used as a string, and if it already ends with a newline, it is taken to be complete, and the result of this function will be the same string. If the message does not end with a newline, then a segment such as "at foo.pl line 37" will be appended, and possibly other clauses indicating the current state of execution. The resulting message will end with a dot and a newline.

Normally, the resulting message is returned in a new mortal SV. During global destruction a single SV may be shared between uses of this function. If "consume" is true, then the function is permitted (but not required) to modify and return "basemsg" instead of allocating a new SV.

```
 SV*  mess_sv(SV* basemsg, bool consume)
```

"pv_display"

Similar to

```
  pv_escape(dsv,pv,cur,pvlim,PERL_PV_ESCAPE_QUOTE);
```

except that an additional "\0" will be appended to the string when len > cur and pv[cur] is "\0".

Note that the final string may be up to 7 chars longer than pvlim.

```
 char*  pv_display(SV *dsv, const char *pv, STRLEN cur, STRLEN len,
                   STRLEN pvlim)
```

"pv_escape"

Escapes at most the first "count" chars of "pv" and puts the results into "dsv" such that the size of the escaped string will not exceed "max" chars and will not contain any incomplete escape sequences. The number of bytes escaped will be returned in the "STRLEN *escaped" parameter if it is not null. When the "dsv" parameter is null no escaping actually occurs, but the number of bytes that would be escaped were it not null will be calculated.

If flags contains "PERL_PV_ESCAPE_QUOTE" then any double quotes in the string will

also be escaped.

Normally the SV will be cleared before the escaped string is prepared, but when "PERL_PV_ESCAPE_NOCLEAR" is set this will not occur.

If "PERL_PV_ESCAPE_UNI" is set then the input string is treated as UTF-8 if "PERL_PV_ESCAPE_UNI_DETECT" is set then the input string is scanned using "is_utf8_string()" to determine if it is UTF-8.

If "PERL_PV_ESCAPE_ALL" is set then all input chars will be output using "\x01F1" style escapes, otherwise if "PERL_PV_ESCAPE_NONASCII" is set, only non-ASCII chars will be escaped using this style; otherwise, only chars above 255 will be so escaped; other non printable chars will use octal or common escaped patterns like "\n". Otherwise, if "PERL_PV_ESCAPE_NOBACKSLASH" then all chars below 255 will be treated as printable and will be output as literals.

If "PERL_PV_ESCAPE_FIRSTCHAR" is set then only the first char of the string will be escaped, regardless of max. If the output is to be in hex, then it will be returned as a plain hex sequence. Thus the output will either be a single char, an octal escape sequence, a special escape like "\n" or a hex value.

If "PERL_PV_ESCAPE_RE" is set then the escape char used will be a "%" and not a "\\". This is because regexes very often contain backslashed sequences, whereas "%" is not a particularly common character in patterns.

Returns a pointer to the escaped text as held by "dsv".

```
 char*  pv_escape(SV *dsv, char const * const str,
                  const STRLEN count, const STRLEN max,
                  STRLEN * const escaped, const U32 flags)
```

"pv_pretty"

Converts a string into something presentable, handling escaping via "pv_escape()" and supporting quoting and ellipses.

If the "PERL_PV_PRETTY_QUOTE" flag is set then the result will be double quoted with any double quotes in the string escaped. Otherwise if the "PERL_PV_PRETTY_LTGT" flag is set then the result be wrapped in angle brackets.

If the "PERL_PV_PRETTY_ELLIPSES" flag is set and not all characters in string were output then an ellipsis "..." will be appended to the string. Note that this happens AFTER it has been quoted.

If "start_color" is non-null then it will be inserted after the opening quote (if

there is one) but before the escaped text. If "end_color" is non-null then it will be inserted after the escaped text but before any quotes or ellipses.

Returns a pointer to the prettified text as held by "dsv".

```
char*  pv_pretty(SV *dsv, char const * const str,
                 const STRLEN count, const STRLEN max,
                 char const * const start_color,
                 char const * const end_color, const U32 flags)
```

"vform"

Like "form" but but the arguments are an encapsulated argument list.

```
char*  vform(const char* pat, va_list* args)
```

"vmess"

"pat" and "args" are a sprintf-style format pattern and encapsulated argument list, respectively. These are used to generate a string message. If the message does not end with a newline, then it will be extended with some indication of the current location in the code, as described for "mess_sv".

Normally, the resulting message is returned in a new mortal SV. During global destruction a single SV may be shared between uses of this function.

```
SV*  vmess(const char* pat, va_list* args)
```

Embedding and Interpreter Cloning

"cv_clone"

Clone a CV, making a lexical closure. "proto" supplies the prototype of the function: its code, pad structure, and other attributes. The prototype is combined with a capture of outer lexicals to which the code refers, which are taken from the currently-executing instance of the immediately surrounding code.

```
CV*  cv_clone(CV* proto)
```

"cv_name"

Returns an SV containing the name of the CV, mainly for use in error reporting. The CV may actually be a GV instead, in which case the returned SV holds the GV's name. Anything other than a GV or CV is treated as a string already holding the sub name, but this could change in the future.

An SV may be passed as a second argument. If so, the name will be assigned to it and it will be returned. Otherwise the returned SV will be a new mortal.

If "flags" has the "CV_NAME_NOTQUAL" bit set, then the package name will not be

included. If the first argument is neither a CV nor a GV, this flag is ignored (subject to change).

SV * cv_name(CV *cv, SV *sv, U32 flags)

"cv_undef"

Clear out all the active components of a CV. This can happen either by an explicit "undef &foo", or by the reference count going to zero. In the former case, we keep the "CvOUTSIDE" pointer, so that any anonymous children can still follow the full lexical scope chain.

void cv_undef(CV* cv)

"find_rundefsv"

Returns the global variable $_.

SV* find_rundefsv()

"find_rundefsvoffset"

"DEPRECATED!" It is planned to remove "find_rundefsvoffset" from a future release of Perl. Do not use it for new code; remove it from existing code.

Until the lexical $_ feature was removed, this function would find the position of the lexical $_ in the pad of the currently-executing function and return the offset in the current pad, or "NOT_IN_PAD".

Now it always returns "NOT_IN_PAD".

PADOFFSET find_rundefsvoffset()

"intro_my"

"Introduce" "my" variables to visible status. This is called during parsing at the end of each statement to make lexical variables visible to subsequent statements.

U32 intro_my()

"load_module"

Loads the module whose name is pointed to by the string part of "name". Note that the actual module name, not its filename, should be given. Eg, "Foo::Bar" instead of "Foo/Bar.pm". ver, if specified and not NULL, provides version semantics similar to "use Foo::Bar VERSION". The optional trailing arguments can be used to specify arguments to the module's "import()" method, similar to "use Foo::Bar VERSION LIST"; their precise handling depends on the flags. The flags argument is a bitwise-ORed collection of any of "PERL_LOADMOD_DENY", "PERL_LOADMOD_NOIMPORT", or "PERL_LOADMOD_IMPORT_OPS" (or 0 for no flags).

If "PERL_LOADMOD_NOIMPORT" is set, the module is loaded as if with an empty import list, as in "use Foo::Bar ()"; this is the only circumstance in which the trailing optional arguments may be omitted entirely. Otherwise, if "PERL_LOADMOD_IMPORT_OPS" is set, the trailing arguments must consist of exactly one "OP*", containing the op tree that produces the relevant import arguments. Otherwise, the trailing arguments must all be "SV*" values that will be used as import arguments; and the list must be terminated with "(SV*) NULL". If neither "PERL_LOADMOD_NOIMPORT" nor "PERL_LOADMOD_IMPORT_OPS" is set, the trailing "NULL" pointer is needed even if no import arguments are desired. The reference count for each specified "SV*" argument is decremented. In addition, the "name" argument is modified.

If "PERL_LOADMOD_DENY" is set, the module is loaded as if with "no" rather than "use".

```
 void  load_module(U32 flags, SV* name, SV* ver, ...)
```

"load_module_nocontext"

Like "load_module" but does not take a thread context ("aTHX") parameter, so is used in situations where the caller doesn't already have the thread context.

```
 void  load_module_nocontext(U32 flags, SV* name, SV* ver, ...)
```

"my_exit"

A wrapper for the C library exit(3), honoring what "PL_exit_flags" in perlapi say to do.

```
 void  my_exit(U32 status)
```

"newPADNAMELIST"

NOTE: "newPADNAMELIST" is experimental and may change or be removed without notice.

Creates a new pad name list. "max" is the highest index for which space is allocated.

```
 PADNAMELIST *  newPADNAMELIST(size_t max)
```

"newPADNAMEouter"

NOTE: "newPADNAMEouter" is experimental and may change or be removed without notice.

Constructs and returns a new pad name. Only use this function for names that refer to outer lexicals. (See also "newPADNAMEpvn".) "outer" is the outer pad name that this one mirrors. The returned pad name has the "PADNAMEt_OUTER" flag already set.

```
 PADNAME *  newPADNAMEouter(PADNAME *outer)
```

"newPADNAMEpvn"

NOTE: "newPADNAMEpvn" is experimental and may change or be removed without notice.

Constructs and returns a new pad name. "s" must be a UTF-8 string. Do not use this

for pad names that point to outer lexicals. See "newPADNAMEouter".

```
PADNAME *  newPADNAMEpvn(const char *s, STRLEN len)
```

"nothreadhook"

Stub that provides thread hook for perl_destruct when there are no threads.

```
int  nothreadhook()
```

"pad_add_anon"

Allocates a place in the currently-compiling pad (via "pad_alloc") for an anonymous function that is lexically scoped inside the currently-compiling function. The function "func" is linked into the pad, and its "CvOUTSIDE" link to the outer scope is weakened to avoid a reference loop.

One reference count is stolen, so you may need to do "SvREFCNT_inc(func)".

"optype" should be an opcode indicating the type of operation that the pad entry is to support. This doesn't affect operational semantics, but is used for debugging.

```
PADOFFSET  pad_add_anon(CV* func, I32 optype)
```

"pad_add_name_pv"

Exactly like "pad_add_name_pvn", but takes a nul-terminated string instead of a string/length pair.

```
PADOFFSET  pad_add_name_pv(const char *name, const U32 flags,
                           HV *typestash, HV *ourstash)
```

"pad_add_name_pvn"

Allocates a place in the currently-compiling pad for a named lexical variable. Stores the name and other metadata in the name part of the pad, and makes preparations to manage the variable's lexical scoping. Returns the offset of the allocated pad slot.

"namepv"/"namelen" specify the variable's name, including leading sigil. If "typestash" is non-null, the name is for a typed lexical, and this identifies the type. If "ourstash" is non-null, it's a lexical reference to a package variable, and this identifies the package. The following flags can be OR'ed together:

```
padadd_OUR          redundantly specifies if it's a package var
padadd_STATE        variable will retain value persistently
padadd_NO_DUP_CHECK skip check for lexical shadowing
```

```
PADOFFSET  pad_add_name_pvn(const char *namepv, STRLEN namelen,
                            U32 flags, HV *typestash,
                            HV *ourstash)
```

"pad_add_name_sv"

Exactly like "pad_add_name_pvn", but takes the name string in the form of an SV instead of a string/length pair.

```
PADOFFSET  pad_add_name_sv(SV *name, U32 flags, HV *typestash,
                           HV *ourstash)
```

"pad_alloc"

NOTE: "pad_alloc" is experimental and may change or be removed without notice.

Allocates a place in the currently-compiling pad, returning the offset of the allocated pad slot. No name is initially attached to the pad slot. "tmptype" is a set of flags indicating the kind of pad entry required, which will be set in the value SV for the allocated pad entry:

SVs_PADMY named lexical variable ("my", "our", "state")

SVs_PADTMP unnamed temporary store

SVf_READONLY constant shared between recursion levels

"SVf_READONLY" has been supported here only since perl 5.20. To work with earlier versions as well, use "SVf_READONLY|SVs_PADTMP". "SVf_READONLY" does not cause the SV in the pad slot to be marked read-only, but simply tells "pad_alloc" that it will be made read-only (by the caller), or at least should be treated as such.

"optype" should be an opcode indicating the type of operation that the pad entry is to support. This doesn't affect operational semantics, but is used for debugging.

```
PADOFFSET  pad_alloc(I32 optype, U32 tmptype)
```

"pad_findmy_pv"

Exactly like "pad_findmy_pvn", but takes a nul-terminated string instead of a string/length pair.

```
PADOFFSET  pad_findmy_pv(const char* name, U32 flags)
```

"pad_findmy_pvn"

Given the name of a lexical variable, find its position in the currently-compiling pad. "namepv"/"namelen" specify the variable's name, including leading sigil. "flags" is reserved and must be zero. If it is not in the current pad but appears in the pad of any lexically enclosing scope, then a pseudo-entry for it is added in the current pad. Returns the offset in the current pad, or "NOT_IN_PAD" if no such lexical is in scope.

```
PADOFFSET  pad_findmy_pvn(const char* namepv, STRLEN namelen,
```

U32 flags)

"pad_findmy_sv"

Exactly like "pad_findmy_pvn", but takes the name string in the form of an SV instead of a string/length pair.

```
PADOFFSET  pad_findmy_sv(SV* name, U32 flags)
```

"padnamelist_fetch"

NOTE: "padnamelist_fetch" is experimental and may change or be removed without notice.

Fetches the pad name from the given index.

```
PADNAME *  padnamelist_fetch(PADNAMELIST *pnl, SSize_t key)
```

"padnamelist_store"

NOTE: "padnamelist_store" is experimental and may change or be removed without notice.

Stores the pad name (which may be null) at the given index, freeing any existing pad name in that slot.

```
PADNAME **  padnamelist_store(PADNAMELIST *pnl, SSize_t key,
                              PADNAME *val)
```

"pad_tidy"

NOTE: "pad_tidy" is experimental and may change or be removed without notice.

Tidy up a pad at the end of compilation of the code to which it belongs. Jobs performed here are: remove most stuff from the pads of anonsub prototypes; give it a @_; mark temporaries as such. "type" indicates the kind of subroutine:

```
padtidy_SUB        ordinary subroutine
padtidy_SUBCLONE   prototype for lexical closure
padtidy_FORMAT     format
```

```
void  pad_tidy(padtidy_type type)
```

"perl_alloc"

Allocates a new Perl interpreter. See perlembed.

```
PerlInterpreter*  perl_alloc()
```

"PERL_ASYNC_CHECK"

Described in perlinterp.

```
void  PERL_ASYNC_CHECK()
```

"perl_clone"

Create and return a new interpreter by cloning the current one.

"perl_clone" takes these flags as parameters:

"CLONEf_COPY_STACKS" - is used to, well, copy the stacks also, without it we only clone the data and zero the stacks, with it we copy the stacks and the new perl interpreter is ready to run at the exact same point as the previous one. The pseudo-fork code uses "COPY_STACKS" while the threads->create doesn't.

"CLONEf_KEEP_PTR_TABLE" - "perl_clone" keeps a ptr_table with the pointer of the old variable as a key and the new variable as a value, this allows it to check if something has been cloned and not clone it again, but rather just use the value and increase the refcount. If "KEEP_PTR_TABLE" is not set then "perl_clone" will kill the ptr_table using the function "ptr_table_free(PL_ptr_table);?PL_ptr_table?=?NULL;". A reason to keep it around is if you want to dup some of your own variables which are outside the graph that perl scans.

"CLONEf_CLONE_HOST" - This is a win32 thing, it is ignored on unix, it tells perl's win32host code (which is c++) to clone itself, this is needed on win32 if you want to run two threads at the same time, if you just want to do some stuff in a separate perl interpreter and then throw it away and return to the original one, you don't need to do anything.

```
 PerlInterpreter*  perl_clone(PerlInterpreter *proto_perl,
                              UV flags)
```

"perl_construct"

Initializes a new Perl interpreter. See perlembed.

```
 void  perl_construct(PerlInterpreter *my_perl)
```

"perl_destruct"

Shuts down a Perl interpreter. See perlembed for a tutorial.

"my_perl" points to the Perl interpreter. It must have been previously created through the use of "perl_alloc" and "perl_construct". It may have been initialised through "perl_parse", and may have been used through "perl_run" and other means. This function should be called for any Perl interpreter that has been constructed with "perl_construct", even if subsequent operations on it failed, for example if "perl_parse" returned a non-zero value.

If the interpreter's "PL_exit_flags" word has the "PERL_EXIT_DESTRUCT_END" flag set, then this function will execute code in "END" blocks before performing the rest of destruction. If it is desired to make any use of the interpreter between "perl_parse" and "perl_destruct" other than just calling "perl_run", then this flag should be set

early on. This matters if "perl_run" will not be called, or if anything else will be done in addition to calling "perl_run".

Returns a value be a suitable value to pass to the C library function "exit" (or to return from "main"), to serve as an exit code indicating the nature of the way the interpreter terminated. This takes into account any failure of "perl_parse" and any early exit from "perl_run". The exit code is of the type required by the host operating system, so because of differing exit code conventions it is not portable to interpret specific numeric values as having specific meanings.

```
int  perl_destruct(PerlInterpreter *my_perl)
```

"perl_free"

Releases a Perl interpreter. See perlembed.

```
void  perl_free(PerlInterpreter *my_perl)
```

"perl_parse"

Tells a Perl interpreter to parse a Perl script. This performs most of the initialisation of a Perl interpreter. See perlembed for a tutorial.

"my_perl" points to the Perl interpreter that is to parse the script. It must have been previously created through the use of "perl_alloc" and "perl_construct".

"xsinit" points to a callback function that will be called to set up the ability for this Perl interpreter to load XS extensions, or may be null to perform no such setup.

"argc" and "argv" supply a set of command-line arguments to the Perl interpreter, as would normally be passed to the "main" function of a C program. "argv[argc]" must be null. These arguments are where the script to parse is specified, either by naming a script file or by providing a script in a "-e" option. If $0 will be written to in the Perl interpreter, then the argument strings must be in writable memory, and so mustn't just be string constants.

"env" specifies a set of environment variables that will be used by this Perl interpreter. If non-null, it must point to a null-terminated array of environment strings. If null, the Perl interpreter will use the environment supplied by the "environ" global variable.

This function initialises the interpreter, and parses and compiles the script specified by the command-line arguments. This includes executing code in "BEGIN", "UNITCHECK", and "CHECK" blocks. It does not execute "INIT" blocks or the main program.

Returns an integer of slightly tricky interpretation. The correct use of the return value is as a truth value indicating whether there was a failure in initialisation. If zero is returned, this indicates that initialisation was successful, and it is safe to proceed to call "perl_run" and make other use of it. If a non-zero value is returned, this indicates some problem that means the interpreter wants to terminate. The interpreter should not be just abandoned upon such failure; the caller should proceed to shut the interpreter down cleanly with "perl_destruct" and free it with "perl_free".

For historical reasons, the non-zero return value also attempts to be a suitable value to pass to the C library function "exit" (or to return from "main"), to serve as an exit code indicating the nature of the way initialisation terminated. However, this isn't portable, due to differing exit code conventions. A historical bug is preserved for the time being: if the Perl built-in "exit" is called during this function's execution, with a type of exit entailing a zero exit code under the host operating system's conventions, then this function returns zero rather than a non-zero value. This bug, [perl #2754], leads to "perl_run" being called (and therefore "INIT" blocks and the main program running) despite a call to "exit". It has been preserved because a popular module-installing module has come to rely on it and needs time to be fixed. This issue is [perl #132577], and the original bug is due to be fixed in Perl 5.30.

```
 int  perl_parse(PerlInterpreter *my_perl, XSINIT_t xsinit,
                 int argc, char** argv, char** env)
```

"perl_run"

Tells a Perl interpreter to run its main program. See perlembed for a tutorial. "my_perl" points to the Perl interpreter. It must have been previously created through the use of "perl_alloc" and "perl_construct", and initialised through "perl_parse". This function should not be called if "perl_parse" returned a non-zero value, indicating a failure in initialisation or compilation.

This function executes code in "INIT" blocks, and then executes the main program. The code to be executed is that established by the prior call to "perl_parse". If the interpreter's "PL_exit_flags" word does not have the "PERL_EXIT_DESTRUCT_END" flag set, then this function will also execute code in "END" blocks. If it is desired to make any further use of the interpreter after calling this function, then "END" blocks should be postponed to "perl_destruct" time by setting that flag.

Returns an integer of slightly tricky interpretation. The correct use of the return value is as a truth value indicating whether the program terminated non-locally. If zero is returned, this indicates that the program ran to completion, and it is safe to make other use of the interpreter (provided that the "PERL_EXIT_DESTRUCT_END" flag was set as described above). If a non-zero value is returned, this indicates that the interpreter wants to terminate early. The interpreter should not be just abandoned because of this desire to terminate; the caller should proceed to shut the interpreter down cleanly with "perl_destruct" and free it with "perl_free".

For historical reasons, the non-zero return value also attempts to be a suitable value to pass to the C library function "exit" (or to return from "main"), to serve as an exit code indicating the nature of the way the program terminated. However, this isn't portable, due to differing exit code conventions. An attempt is made to return an exit code of the type required by the host operating system, but because it is constrained to be non-zero, it is not necessarily possible to indicate every type of exit. It is only reliable on Unix, where a zero exit code can be augmented with a set bit that will be ignored. In any case, this function is not the correct place to acquire an exit code: one should get that from "perl_destruct".

```
 int  perl_run(PerlInterpreter *my_perl)
```

"PERL_SYS_INIT"

Provides system-specific tune up of the C runtime environment necessary to run Perl interpreters. This should be called only once, before creating any Perl interpreters.

```
 void  PERL_SYS_INIT(int *argc, char*** argv)
```

"PERL_SYS_INIT3"

Provides system-specific tune up of the C runtime environment necessary to run Perl interpreters. This should be called only once, before creating any Perl interpreters.

```
 void  PERL_SYS_INIT3(int *argc, char*** argv, char*** env)
```

"PERL_SYS_TERM"

Provides system-specific clean up of the C runtime environment after running Perl interpreters. This should be called only once, after freeing any remaining Perl interpreters.

```
 void  PERL_SYS_TERM()
```

"PL_exit_flags"

Contains flags controlling perl's behaviour on exit():

? "PERL_EXIT_DESTRUCT_END"

If set, END blocks are executed when the interpreter is destroyed. This is

normally set by perl itself after the interpreter is constructed.

? "PERL_EXIT_ABORT"

Call "abort()" on exit. This is used internally by perl itself to abort if exit

is called while processing exit.

? "PERL_EXIT_WARN"

Warn on exit.

? "PERL_EXIT_EXPECTED"

Set by the "exit" in perlfunc operator.

U8 PL_exit_flags

"PL_perl_destruct_level"

This value may be set when embedding for full cleanup.

Possible values:

? 0 - none

? 1 - full

? 2 or greater - full with checks.

If $ENV{PERL_DESTRUCT_LEVEL} is set to an integer greater than the value of

"PL_perl_destruct_level" its value is used instead.

On threaded perls, each thread has an independent copy of this variable; each

initialized at creation time with the current value of the creating thread's copy.

signed char PL_perl_destruct_level

"require_pv"

Tells Perl to "require" the file named by the string argument. It is analogous to the

Perl code "eval "require '$file'"". It's even implemented that way; consider using

load_module instead.

NOTE: the "perl_require_pv()" form is deprecated.

void require_pv(const char* pv)

"UVf"

"DEPRECATED!" It is planned to remove "UVf" from a future release of Perl. Do not

use it for new code; remove it from existing code.

Obsolete form of "UVuf", which you should convert to instead use

const char * UVf

"vload_module"

Like "load_module" but the arguments are an encapsulated argument list.

```
void  vload_module(U32 flags, SV* name, SV* ver, va_list* args)
```

Errno

"sv_string_from_errnum"

Generates the message string describing an OS error and returns it as an SV. "errnum" must be a value that "errno" could take, identifying the type of error.

If "tgtsv" is non-null then the string will be written into that SV (overwriting existing content) and it will be returned. If "tgtsv" is a null pointer then the string will be written into a new mortal SV which will be returned.

The message will be taken from whatever locale would be used by $!, and will be encoded in the SV in whatever manner would be used by $!. The details of this process are subject to future change. Currently, the message is taken from the C locale by default (usually producing an English message), and from the currently selected locale when in the scope of the "use locale" pragma. A heuristic attempt is made to decode the message from the locale's character encoding, but it will only be decoded as either UTF-8 or ISO-8859-1. It is always correctly decoded in a UTF-8 locale, usually in an ISO-8859-1 locale, and never in any other locale.

The SV is always returned containing an actual string, and with no other OK bits set.

Unlike $!, a message is even yielded for "errnum" zero (meaning success), and if no useful message is available then a useless string (currently empty) is returned.

```
SV*  sv_string_from_errnum(int errnum, SV* tgtsv)
```

Exception Handling (simple) Macros

"dXCPT"

Set up necessary local variables for exception handling. See "Exception Handling" in perlguts.

```
  dXCPT;
```

"JMPENV_JUMP"

Described in perlinterp.

```
void  JMPENV_JUMP(int v)
```

"JMPENV_PUSH"

Described in perlinterp.

```
void  JMPENV_PUSH(int v)
```

"PL_restartop"

Described in perlinterp.

"XCPT_CATCH"

Introduces a catch block. See "Exception Handling" in perlguts.

"XCPT_RETHROW"

Rethrows a previously caught exception. See "Exception Handling" in perlguts.

```
XCPT_RETHROW;
```

"XCPT_TRY_END"

Ends a try block. See "Exception Handling" in perlguts.

"XCPT_TRY_START"

Starts a try block. See "Exception Handling" in perlguts.

Filesystem configuration values

Also see "List of capability HAS_foo symbols".

"DIRNAMLEN"

This symbol, if defined, indicates to the C program that the length of directory entry names is provided by a "d_namlen" field. Otherwise you need to do "strlen()" on the "d_name" field.

"DOSUID"

This symbol, if defined, indicates that the C program should check the script that it is executing for setuid/setgid bits, and attempt to emulate setuid/setgid on systems that have disabled setuid #! scripts because the kernel can't do it securely. It is up to the package designer to make sure that this emulation is done securely. Among other things, it should do an fstat on the script it just opened to make sure it really is a setuid/setgid script, it should make sure the arguments passed correspond exactly to the argument on the #! line, and it should not trust any subprocesses to which it must pass the filename rather than the file descriptor of the script to be executed.

"EOF_NONBLOCK"

This symbol, if defined, indicates to the C program that a "read()" on a non-blocking file descriptor will return 0 on "EOF", and not the value held in "RD_NODATA" (-1 usually, in that case!).

"FCNTL_CAN_LOCK"

This symbol, if defined, indicates that "fcntl()" can be used for file locking.

Normally on Unix systems this is defined. It may be undefined on "VMS".

"FFLUSH_ALL"

This symbol, if defined, tells that to flush all pending stdio output one must loop through all the stdio file handles stored in an array and fflush them. Note that if "fflushNULL" is defined, fflushall will not even be probed for and will be left undefined.

"FFLUSH_NULL"

This symbol, if defined, tells that "fflush(NULL)" correctly flushes all pending stdio output without side effects. In particular, on some platforms calling "fflush(NULL)" *still* corrupts "STDIN" if it is a pipe.

"FILE_base"

This macro is used to access the "_base" field (or equivalent) of the "FILE" structure pointed to by its argument. This macro will always be defined if "USE_STDIO_BASE" is defined.

```
 void *  FILE_base(FILE * f)
```

"FILE_bufsiz"

This macro is used to determine the number of bytes in the I/O buffer pointed to by "_base" field (or equivalent) of the "FILE" structure pointed to its argument. This macro will always be defined if "USE_STDIO_BASE" is defined.

```
 Size_t  FILE_bufsiz(FILE *f)
```

"FILE_cnt"

This macro is used to access the "_cnt" field (or equivalent) of the "FILE" structure pointed to by its argument. This macro will always be defined if "USE_STDIO_PTR" is defined.

```
 Size_t  FILE_cnt(FILE * f)
```

"FILE_ptr"

This macro is used to access the "_ptr" field (or equivalent) of the "FILE" structure pointed to by its argument. This macro will always be defined if "USE_STDIO_PTR" is defined.

```
 void *  FILE_ptr(FILE * f)
```

"FLEXFILENAMES"

This symbol, if defined, indicates that the system supports filenames longer than 14 characters.

"HAS_DIR_DD_FD"

This symbol, if defined, indicates that the the "DIR"* dirstream structure contains a member variable named "dd_fd".

"HAS_DUP2"

This symbol, if defined, indicates that the "dup2" routine is available to duplicate file descriptors.

"HAS_DUP3"

This symbol, if defined, indicates that the "dup3" routine is available to duplicate file descriptors.

"HAS_FAST_STDIO"

This symbol, if defined, indicates that the "fast stdio" is available to manipulate the stdio buffers directly.

"HAS_FCHDIR"

This symbol, if defined, indicates that the "fchdir" routine is available to change directory using a file descriptor.

"HAS_FCNTL"

This symbol, if defined, indicates to the C program that the "fcntl()" function exists.

"HAS_FDCLOSE"

This symbol, if defined, indicates that the "fdclose" routine is available to free a "FILE" structure without closing the underlying file descriptor. This function appeared in "FreeBSD" 10.2.

"HAS_FPATHCONF"

This symbol, if defined, indicates that "pathconf()" is available to determine file-system related limits and options associated with a given open file descriptor.

"HAS_FPOS64_T"

This symbol will be defined if the C compiler supports "fpos64_t".

"HAS_FSTATFS"

This symbol, if defined, indicates that the "fstatfs" routine is available to stat filesystems by file descriptors.

"HAS_FSTATVFS"

This symbol, if defined, indicates that the "fstatvfs" routine is available to stat filesystems by file descriptors.

"HAS_GETFSSTAT"

This symbol, if defined, indicates that the "getfsstat" routine is available to stat filesystems in bulk.

"HAS_GETMNT"

This symbol, if defined, indicates that the "getmnt" routine is available to get filesystem mount info by filename.

"HAS_GETMNTENT"

This symbol, if defined, indicates that the "getmntent" routine is available to iterate through mounted file systems to get their info.

"HAS_HASMNTOPT"

This symbol, if defined, indicates that the "hasmntopt" routine is available to query the mount options of file systems.

"HAS_LSEEK_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "lseek()" function. Otherwise, it is up to the program to supply one. A good guess is

```
extern off_t lseek(int, off_t, int);
```

"HAS_MKDIR"

This symbol, if defined, indicates that the "mkdir" routine is available to create directories. Otherwise you should fork off a new process to exec /bin/mkdir.

"HAS_OFF64_T"

This symbol will be defined if the C compiler supports "off64_t".

"HAS_OPEN3"

This manifest constant lets the C program know that the three argument form of open(2) is available.

"HAS_OPENAT"

This symbol is defined if the "openat()" routine is available.

"HAS_POLL"

This symbol, if defined, indicates that the "poll" routine is available to "poll" active file descriptors. Please check "I_POLL" and "I_SYS_POLL" to know which header should be included as well.

"HAS_READDIR"

This symbol, if defined, indicates that the "readdir" routine is available to read

directory entries. You may have to include dirent.h. See "I_DIRENT".

"HAS_READDIR64_R"

This symbol, if defined, indicates that the "readdir64_r" routine is available to readdir64 re-entrantly.

"HAS_REWINDDIR"

This symbol, if defined, indicates that the "rewinddir" routine is available. You may have to include dirent.h. See "I_DIRENT".

"HAS_RMDIR"

This symbol, if defined, indicates that the "rmdir" routine is available to remove directories. Otherwise you should fork off a new process to exec /bin/rmdir.

"HAS_SEEKDIR"

This symbol, if defined, indicates that the "seekdir" routine is available. You may have to include dirent.h. See "I_DIRENT".

"HAS_SELECT"

This symbol, if defined, indicates that the "select" routine is available to "select" active file descriptors. If the timeout field is used, sys/time.h may need to be included.

"HAS_SETVBUF"

This symbol, if defined, indicates that the "setvbuf" routine is available to change buffering on an open stdio stream. to a line-buffered mode.

"HAS_STDIO_STREAM_ARRAY"

This symbol, if defined, tells that there is an array holding the stdio streams.

"HAS_STRUCT_FS_DATA"

This symbol, if defined, indicates that the "struct fs_data" to do "statfs()" is supported.

"HAS_STRUCT_STATFS"

This symbol, if defined, indicates that the "struct statfs" to do "statfs()" is supported.

"HAS_STRUCT_STATFS_F_FLAGS"

This symbol, if defined, indicates that the "struct statfs" does have the "f_flags" member containing the mount flags of the filesystem containing the file. This kind of "struct statfs" is coming from sys/mount.h ("BSD" 4.3), not from sys/statfs.h ("SYSV"). Older "BSDs" (like Ultrix) do not have "statfs()" and "struct statfs", they

have "ustat()" and "getmnt()" with "struct ustat" and "struct fs_data".

"HAS_TELLDIR"

This symbol, if defined, indicates that the "telldir" routine is available. You may have to include dirent.h. See "I_DIRENT".

"HAS_USTAT"

This symbol, if defined, indicates that the "ustat" system call is available to query file system statistics by "dev_t".

"I_FCNTL"

This manifest constant tells the C program to include fcntl.h.

```
#ifdef I_FCNTL
    #include <fcntl.h>
#endif
```

"I_SYS_DIR"

This symbol, if defined, indicates to the C program that it should include sys/dir.h.

```
#ifdef I_SYS_DIR
    #include <sys_dir.h>
#endif
```

"I_SYS_FILE"

This symbol, if defined, indicates to the C program that it should include sys/file.h to get definition of "R_OK" and friends.

```
#ifdef I_SYS_FILE
    #include <sys_file.h>
#endif
```

"I_SYS_NDIR"

This symbol, if defined, indicates to the C program that it should include sys/ndir.h.

```
#ifdef I_SYS_NDIR
    #include <sys_ndir.h>
#endif
```

"I_SYS_STATFS"

This symbol, if defined, indicates that sys/statfs.h exists.

```
#ifdef I_SYS_STATFS
    #include <sys_statfs.h>
#endif
```

"LSEEKSIZE"

This symbol holds the number of bytes used by the "Off_t".

"RD_NODATA"

This symbol holds the return code from "read()" when no data is present on the non-blocking file descriptor. Be careful! If "EOF_NONBLOCK" is not defined, then you can't distinguish between no data and "EOF" by issuing a "read()". You'll have to find another way to tell for sure!

"READDIR64_R_PROTO"

This symbol encodes the prototype of "readdir64_r". It is zero if "d_readdir64_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_readdir64_r" is defined.

"STDCHAR"

This symbol is defined to be the type of char used in stdio.h. It has the values "unsigned char" or "char".

"STDIO_CNT_LVALUE"

This symbol is defined if the "FILE_cnt" macro can be used as an lvalue.

"STDIO_PTR_LVALUE"

This symbol is defined if the "FILE_ptr" macro can be used as an lvalue.

"STDIO_PTR_LVAL_NOCHANGE_CNT"

This symbol is defined if using the "FILE_ptr" macro as an lvalue to increase the pointer by n leaves "File_cnt(fp)" unchanged.

"STDIO_PTR_LVAL_SETS_CNT"

This symbol is defined if using the "FILE_ptr" macro as an lvalue to increase the pointer by n has the side effect of decreasing the value of "File_cnt(fp)" by n.

"STDIO_STREAM_ARRAY"

This symbol tells the name of the array holding the stdio streams. Usual values include "_iob", "__iob", and "__sF".

"ST_INO_SIGN"

This symbol holds the signedness of "struct stat"'s "st_ino". 1 for unsigned, -1 for signed.

"ST_INO_SIZE"

This variable contains the size of "struct stat"'s "st_ino" in bytes.

"VAL_EAGAIN"

This symbol holds the errno error code set by "read()" when no data was present on the non-blocking file descriptor.

"VAL_O_NONBLOCK"

This symbol is to be used during "open()" or "fcntl(F_SETFL)" to turn on non-blocking I/O for the file descriptor. Note that there is no way back, i.e. you cannot turn it blocking again this way. If you wish to alternatively switch between blocking and non-blocking, use the "ioctl(FIOSNBIO)" call instead, but that is not supported by all devices.

"VOID_CLOSEDIR"

This symbol, if defined, indicates that the "closedir()" routine does not return a value.

Floating point configuration values

Also "List of capability HAS_foo symbols" lists capabilities that arent in this section.

For example "HAS_ASINH", for the hyperbolic sine function.

"CASTFLAGS"

This symbol contains flags that say what difficulties the compiler has casting odd floating values to unsigned long:

0 = ok

1 = couldn't cast < 0

2 = couldn't cast >= 0x80000000

4 = couldn't cast in argument expression list

"CASTNEGFLOAT"

This symbol is defined if the C compiler can cast negative numbers to unsigned longs, ints and shorts.

"DOUBLE_HAS_INF"

This symbol, if defined, indicates that the double has the infinity.

"DOUBLE_HAS_NAN"

This symbol, if defined, indicates that the double has the not-a-number.

"DOUBLE_HAS_NEGATIVE_ZERO"

This symbol, if defined, indicates that the double has the "negative_zero".

"DOUBLE_HAS_SUBNORMALS"

This symbol, if defined, indicates that the double has the subnormals (denormals).

"DOUBLEINFBYTES"

This symbol, if defined, is a comma-separated list of hexadecimal bytes for the double precision infinity.

"DOUBLEKIND"

"DOUBLEKIND" will be one of "DOUBLE_IS_IEEE_754_32_BIT_LITTLE_ENDIAN" "DOUBLE_IS_IEEE_754_32_BIT_BIG_ENDIAN" "DOUBLE_IS_IEEE_754_64_BIT_LITTLE_ENDIAN" "DOUBLE_IS_IEEE_754_64_BIT_BIG_ENDIAN" "DOUBLE_IS_IEEE_754_128_BIT_LITTLE_ENDIAN" "DOUBLE_IS_IEEE_754_128_BIT_BIG_ENDIAN" "DOUBLE_IS_IEEE_754_64_BIT_MIXED_ENDIAN_LE_BE" "DOUBLE_IS_IEEE_754_64_BIT_MIXED_ENDIAN_BE_LE" "DOUBLE_IS_VAX_F_FLOAT" "DOUBLE_IS_VAX_D_FLOAT" "DOUBLE_IS_VAX_G_FLOAT" "DOUBLE_IS_IBM_SINGLE_32_BIT" "DOUBLE_IS_IBM_DOUBLE_64_BIT" "DOUBLE_IS_CRAY_SINGLE_64_BIT" "DOUBLE_IS_UNKNOWN_FORMAT"

"DOUBLEMANTBITS"

This symbol, if defined, tells how many mantissa bits there are in double precision floating point format. Note that this is usually "DBL_MANT_DIG" minus one, since with the standard "IEEE" 754 formats "DBL_MANT_DIG" includes the implicit bit, which doesn't really exist.

"DOUBLENANBYTES"

This symbol, if defined, is a comma-separated list of hexadecimal bytes (0xHH) for the double precision not-a-number.

"DOUBLESIZE"

This symbol contains the size of a double, so that the C preprocessor can make decisions based on it.

"DOUBLE_STYLE_CRAY"

This symbol, if defined, indicates that the double is the 64-bit "CRAY" mainframe format.

"DOUBLE_STYLE_IBM"

This symbol, if defined, indicates that the double is the 64-bit "IBM" mainframe format.

"DOUBLE_STYLE_IEEE"

This symbol, if defined, indicates that the double is the 64-bit "IEEE" 754.

"DOUBLE_STYLE_VAX"

This symbol, if defined, indicates that the double is the 64-bit "VAX" format D or G.

"HAS_ATOLF"

This symbol, if defined, indicates that the "atolf" routine is available to convert strings into long doubles.

"HAS_CLASS"

This symbol, if defined, indicates that the "class" routine is available to classify doubles. Available for example in "AIX". The returned values are defined in float.h and are:

FP_PLUS_NORM Positive normalized, nonzero

FP_MINUS_NORM Negative normalized, nonzero

FP_PLUS_DENORM Positive denormalized, nonzero

FP_MINUS_DENORM Negative denormalized, nonzero

FP_PLUS_ZERO +0.0

FP_MINUS_ZERO -0.0

FP_PLUS_INF +INF

FP_MINUS_INF -INF

FP_NANS Signaling Not a Number (NaNS)

FP_NANQ Quiet Not a Number (NaNQ)

"HAS_FINITE"

This symbol, if defined, indicates that the "finite" routine is available to check whether a double is "finite" (non-infinity non-NaN).

"HAS_FINITEL"

This symbol, if defined, indicates that the "finitel" routine is available to check whether a long double is finite (non-infinity non-NaN).

"HAS_FPCLASS"

This symbol, if defined, indicates that the "fpclass" routine is available to classify doubles. Available for example in Solaris/"SVR4". The returned values are defined in ieeefp.h and are:

FP_SNAN signaling NaN

FP_QNAN quiet NaN

FP_NINF negative infinity

FP_PINF positive infinity

FP_NDENORM negative denormalized non-zero

FP_PDENORM positive denormalized non-zero

FP_NZERO negative zero

FP_PZERO positive zero

FP_NNORM negative normalized non-zero

FP_PNORM positive normalized non-zero

"HAS_FPCLASSIFY"

This symbol, if defined, indicates that the "fpclassify" routine is available to classify doubles. Available for example in HP-UX. The returned values are defined in math.h and are

FP_NORMAL Normalized

FP_ZERO Zero

FP_INFINITE Infinity

FP_SUBNORMAL Denormalized

FP_NAN NaN

"HAS_FPCLASSL"

This symbol, if defined, indicates that the "fpclassl" routine is available to classify long doubles. Available for example in "IRIX". The returned values are defined in ieeefp.h and are:

FP_SNAN signaling NaN

FP_QNAN quiet NaN

FP_NINF negative infinity

FP_PINF positive infinity

FP_NDENORM negative denormalized non-zero

FP_PDENORM positive denormalized non-zero

FP_NZERO negative zero

FP_PZERO positive zero

FP_NNORM negative normalized non-zero

FP_PNORM positive normalized non-zero

"HAS_FPGETROUND"

This symbol, if defined, indicates that the "fpgetround" routine is available to get the floating point rounding mode.

"HAS_FP_CLASS"

This symbol, if defined, indicates that the "fp_class" routine is available to classify doubles. Available for example in Digital "UNIX". The returned values are defined in math.h and are:

FP_SNAN Signaling NaN (Not-a-Number)

FP_QNAN Quiet NaN (Not-a-Number)

FP_POS_INF +infinity

FP_NEG_INF -infinity

FP_POS_NORM Positive normalized

FP_NEG_NORM Negative normalized

FP_POS_DENORM Positive denormalized

FP_NEG_DENORM Negative denormalized

FP_POS_ZERO +0.0 (positive zero)

FP_NEG_ZERO -0.0 (negative zero)

"HAS_FP_CLASSIFY"

This symbol, if defined, indicates that the "fp_classify" routine is available to classify doubles. The values are defined in math.h

FP_NORMAL Normalized

FP_ZERO Zero

FP_INFINITE Infinity

FP_SUBNORMAL Denormalized

FP_NAN NaN

"HAS_FP_CLASSL"

This symbol, if defined, indicates that the "fp_classl" routine is available to classify long doubles. Available for example in Digital "UNIX". See for possible values "HAS_FP_CLASS".

"HAS_FREXPL"

This symbol, if defined, indicates that the "frexpl" routine is available to break a long double floating-point number into a normalized fraction and an integral power of 2.

"HAS_ILOGB"

This symbol, if defined, indicates that the "ilogb" routine is available to get integer exponent of a floating-point value.

"HAS_ISFINITE"

This symbol, if defined, indicates that the "isfinite" routine is available to check whether a double is finite (non-infinity non-NaN).

"HAS_ISFINITEL"

This symbol, if defined, indicates that the "isfinitel" routine is available to check whether a long double is finite. (non-infinity non-NaN).

"HAS_ISINF"

This symbol, if defined, indicates that the "isinf" routine is available to check whether a double is an infinity.

"HAS_ISINFL"

This symbol, if defined, indicates that the "isinfl" routine is available to check whether a long double is an infinity.

"HAS_ISNAN"

This symbol, if defined, indicates that the "isnan" routine is available to check whether a double is a NaN.

"HAS_ISNANL"

This symbol, if defined, indicates that the "isnanl" routine is available to check whether a long double is a NaN.

"HAS_ISNORMAL"

This symbol, if defined, indicates that the "isnormal" routine is available to check whether a double is normal (non-zero normalized).

"HAS_J0"

This symbol, if defined, indicates to the C program that the "j0()" function is available for Bessel functions of the first kind of the order zero, for doubles.

"HAS_J0L"

This symbol, if defined, indicates to the C program that the "j0l()" function is available for Bessel functions of the first kind of the order zero, for long doubles.

"HAS_LDBL_DIG"

This symbol, if defined, indicates that this system's float.h or limits.h defines the symbol "LDBL_DIG", which is the number of significant digits in a long double precision number. Unlike for "DBL_DIG", there's no good guess for "LDBL_DIG" if it is undefined.

"HAS_LDEXPL"

This symbol, if defined, indicates that the "ldexpl" routine is available to shift a long double floating-point number by an integral power of 2.

"HAS_LLRINT"

This symbol, if defined, indicates that the "llrint" routine is available to return

the long long value closest to a double (according to the current rounding mode).

"HAS_LLRINTL"

This symbol, if defined, indicates that the "llrintl" routine is available to return the long long value closest to a long double (according to the current rounding mode).

"HAS_LLROUNDL"

This symbol, if defined, indicates that the "llroundl" routine is available to return the nearest long long value away from zero of the long double argument value.

"HAS_LONG_DOUBLE"

This symbol will be defined if the C compiler supports long doubles.

"HAS_LRINT"

This symbol, if defined, indicates that the "lrint" routine is available to return the integral value closest to a double (according to the current rounding mode).

"HAS_LRINTL"

This symbol, if defined, indicates that the "lrintl" routine is available to return the integral value closest to a long double (according to the current rounding mode).

"HAS_LROUNDL"

This symbol, if defined, indicates that the "lroundl" routine is available to return the nearest integral value away from zero of the long double argument value.

"HAS_MODFL"

This symbol, if defined, indicates that the "modfl" routine is available to split a long double x into a fractional part f and an integer part i such that |f| < 1.0 and (f + i) = x.

"HAS_NAN"

This symbol, if defined, indicates that the "nan" routine is available to generate NaN.

"HAS_NEXTTOWARD"

This symbol, if defined, indicates that the "nexttoward" routine is available to return the next machine representable long double from x in direction y.

"HAS_REMAINDER"

This symbol, if defined, indicates that the "remainder" routine is available to return the floating-point "remainder".

"HAS_SCALBN"

This symbol, if defined, indicates that the "scalbn" routine is available to multiply

floating-point number by integral power of radix.

"HAS_SIGNBIT"

This symbol, if defined, indicates that the "signbit" routine is available to check if the given number has the sign bit set. This should include correct testing of -0.0. This will only be set if the "signbit()" routine is safe to use with the NV type used internally in perl. Users should call "Perl_signbit()", which will be #defined to the system's "signbit()" function or macro if this symbol is defined.

"HAS_SQRTL"

This symbol, if defined, indicates that the "sqrtl" routine is available to do long double square roots.

"HAS_STRTOD_L"

This symbol, if defined, indicates that the "strtod_l" routine is available to convert strings to long doubles.

"HAS_STRTOLD"

This symbol, if defined, indicates that the "strtold" routine is available to convert strings to long doubles.

"HAS_STRTOLD_L"

This symbol, if defined, indicates that the "strtold_l" routine is available to convert strings to long doubles.

"HAS_TRUNC"

This symbol, if defined, indicates that the "trunc" routine is available to round doubles towards zero.

"HAS_UNORDERED"

This symbol, if defined, indicates that the "unordered" routine is available to check whether two doubles are "unordered" (effectively: whether either of them is NaN)

"I_FENV"

This symbol, if defined, indicates to the C program that it should include fenv.h to get the floating point environment definitions.

```
#ifdef I_FENV
    #include <fenv.h>
#endif
```

"I_QUADMATH"

This symbol, if defined, indicates that quadmath.h exists and should be included.

```
#ifdef I_QUADMATH
    #include <quadmath.h>
#endif
```

"LONGDBLINFBYTES"

This symbol, if defined, is a comma-separated list of hexadecimal bytes for the long double precision infinity.

"LONGDBLMANTBITS"

This symbol, if defined, tells how many mantissa bits there are in long double precision floating point format. Note that this can be "LDBL_MANT_DIG" minus one, since "LDBL_MANT_DIG" can include the "IEEE" 754 implicit bit. The common x86-style 80-bit long double does not have an implicit bit.

"LONGDBLNANBYTES"

This symbol, if defined, is a comma-separated list of hexadecimal bytes (0xHH) for the long double precision not-a-number.

"LONG_DOUBLEKIND"

"LONG_DOUBLEKIND" will be one of "LONG_DOUBLE_IS_DOUBLE" "LONG_DOUBLE_IS_IEEE_754_128_BIT_LITTLE_ENDIAN" "LONG_DOUBLE_IS_IEEE_754_128_BIT_BIG_ENDIAN" "LONG_DOUBLE_IS_X86_80_BIT_LITTLE_ENDIAN" "LONG_DOUBLE_IS_X86_80_BIT_BIG_ENDIAN" "LONG_DOUBLE_IS_DOUBLEDOUBLE_128_BIT_LE_LE" "LONG_DOUBLE_IS_DOUBLEDOUBLE_128_BIT_BE_BE" "LONG_DOUBLE_IS_DOUBLEDOUBLE_128_BIT_LE_BE" "LONG_DOUBLE_IS_DOUBLEDOUBLE_128_BIT_BE_LE" "LONG_DOUBLE_IS_DOUBLEDOUBLE_128_BIT_LITTLE_ENDIAN" "LONG_DOUBLE_IS_DOUBLEDOUBLE_128_BIT_BIG_ENDIAN" "LONG_DOUBLE_IS_VAX_H_FLOAT" "LONG_DOUBLE_IS_UNKNOWN_FORMAT" It is only defined if the system supports long doubles.

"LONG_DOUBLESIZE"

This symbol contains the size of a long double, so that the C preprocessor can make decisions based on it. It is only defined if the system supports long doubles. Note that this is "sizeof(long double)", which may include unused bytes.

"LONG_DOUBLE_STYLE_IEEE"

This symbol, if defined, indicates that the long double is any of the "IEEE" 754 style long doubles: "LONG_DOUBLE_STYLE_IEEE_STD", "LONG_DOUBLE_STYLE_IEEE_EXTENDED",

"LONG_DOUBLE_STYLE_IEEE_DOUBLEDOUBLE".

"LONG_DOUBLE_STYLE_IEEE_DOUBLEDOUBLE"

This symbol, if defined, indicates that the long double is the 128-bit double-double.

"LONG_DOUBLE_STYLE_IEEE_EXTENDED"

This symbol, if defined, indicates that the long double is the 80-bit "IEEE" 754. Note that despite the 'extended' this is less than the 'std', since this is an extension of the double precision.

"LONG_DOUBLE_STYLE_IEEE_STD"

This symbol, if defined, indicates that the long double is the 128-bit "IEEE" 754.

"LONG_DOUBLE_STYLE_VAX"

This symbol, if defined, indicates that the long double is the 128-bit "VAX" format H.

"NVMANTBITS"

This symbol, if defined, tells how many mantissa bits (not including implicit bit) there are in a Perl NV. This depends on which floating point type was chosen.

"NV_OVERFLOWS_INTEGERS_AT"

This symbol gives the largest integer value that NVs can hold. This value + 1.0 cannot be stored accurately. It is expressed as constant floating point expression to reduce the chance of decimal/binary conversion issues. If it can not be determined, the value 0 is given.

"NV_PRESERVES_UV"

This symbol, if defined, indicates that a variable of type "NVTYPE" can preserve all the bits of a variable of type "UVTYPE".

"NV_PRESERVES_UV_BITS"

This symbol contains the number of bits a variable of type "NVTYPE" can preserve of a variable of type "UVTYPE".

"NVSIZE"

This symbol contains the "sizeof(NV)". Note that some floating point formats have unused bytes. The most notable example is the x86* 80-bit extended precision which comes in byte sizes of 12 and 16 (for 32 and 64 bit platforms, respectively), but which only uses 10 bytes. Perl compiled with "-Duselongdouble" on x86* is like this.

"NVTYPE"

This symbol defines the C type used for Perl's NV.

"NV_ZERO_IS_ALLBITS_ZERO"

This symbol, if defined, indicates that a variable of type "NVTYPE" stores 0.0 in memory as all bits zero.

Formats

These are used for formatting the corresponding type For example, instead of saying

```
 Perl_newSVpvf(pTHX_ "Create an SV with a %d in it\n", iv);
```

use

```
 Perl_newSVpvf(pTHX_ "Create an SV with a " IVdf " in it\n", iv);
```

This keeps you from having to know if, say an IV, needs to be printed as %d, %ld, or something else.

"IVdf"

This symbol defines the format string used for printing a Perl IV as a signed decimal integer.

"NVef"

This symbol defines the format string used for printing a Perl NV using %e-ish floating point format.

"NVff"

This symbol defines the format string used for printing a Perl NV using %f-ish floating point format.

"NVgf"

This symbol defines the format string used for printing a Perl NV using %g-ish floating point format.

"PERL_PRIeldbl"

This symbol, if defined, contains the string used by stdio to format long doubles (format 'e') for output.

"PERL_PRIfldbl"

This symbol, if defined, contains the string used by stdio to format long doubles (format 'f') for output.

"PERL_PRIgldbl"

This symbol, if defined, contains the string used by stdio to format long doubles (format 'g') for output.

"PERL_SCNfldbl"

This symbol, if defined, contains the string used by stdio to format long doubles (format 'f') for input.

"PRINTF_FORMAT_NULL_OK"

Allows "__printf__" format to be null when checking printf-style

"UTF8f"

Described in perlguts.

"UTF8fARG"

Described in perlguts.

UTF8fARG(bool is_utf8, Size_t byte_len, char *str)

"UVof"

This symbol defines the format string used for printing a Perl UV as an unsigned octal integer.

"UVuf"

This symbol defines the format string used for printing a Perl UV as an unsigned decimal integer.

"UVXf"

This symbol defines the format string used for printing a Perl UV as an unsigned hexadecimal integer in uppercase "ABCDEF".

"UVxf"

This symbol defines the format string used for printing a Perl UV as an unsigned hexadecimal integer in lowercase abcdef.

General Configuration

This section contains configuration information not otherwise found in the more specialized sections of this document. At the end is a list of "#defines" whose name should be enough to tell you what they do, and a list of #defines which tell you if you need to "#include" files to get the corresponding functionality.

"BYTEORDER"

This symbol holds the hexadecimal constant defined in byteorder, in a UV, i.e. 0x1234 or 0x4321 or 0x12345678, etc... If the compiler supports cross-compiling or multiple-architecture binaries, use compiler-defined macros to determine the byte order.

"CHARBITS"

This symbol contains the size of a char, so that the C preprocessor can make decisions based on it.

"DB_VERSION_MAJOR_CFG"

This symbol, if defined, defines the major version number of Berkeley DB found in the

db.h header when Perl was configured.

"DB_VERSION_MINOR_CFG"

This symbol, if defined, defines the minor version number of Berkeley DB found in the db.h header when Perl was configured. For DB version 1 this is always 0.

"DB_VERSION_PATCH_CFG"

This symbol, if defined, defines the patch version number of Berkeley DB found in the db.h header when Perl was configured. For DB version 1 this is always 0.

"DEFAULT_INC_EXCLUDES_DOT"

This symbol, if defined, removes the legacy default behavior of including '.' at the end of @"INC".

"DLSYM_NEEDS_UNDERSCORE"

This symbol, if defined, indicates that we need to prepend an underscore to the symbol name before calling "dlsym()". This only makes sense if you *have* dlsym, which we will presume is the case if you're using dl_dlopen.xs.

"EBCDIC"

This symbol, if defined, indicates that this system uses "EBCDIC" encoding.

"HAS_CSH"

This symbol, if defined, indicates that the C-shell exists.

"HAS_GETHOSTNAME"

This symbol, if defined, indicates that the C program may use the "gethostname()" routine to derive the host name. See also "HAS_UNAME" and "PHOSTNAME".

"HAS_GNULIBC"

This symbol, if defined, indicates to the C program that the "GNU" C library is being used. A better check is to use the "__GLIBC__" and "__GLIBC_MINOR__" symbols supplied with glibc.

"HAS_LGAMMA"

This symbol, if defined, indicates that the "lgamma" routine is available to do the log gamma function. See also "HAS_TGAMMA" and "HAS_LGAMMA_R".

"HAS_LGAMMA_R"

This symbol, if defined, indicates that the "lgamma_r" routine is available to do the log gamma function without using the global signgam variable.

"HAS_PRCTL_SET_NAME"

This symbol, if defined, indicates that the prctl routine is available to set process

title and supports "PR_SET_NAME".

"HAS_PROCSELFEXE"

This symbol is defined if "PROCSELFEXE_PATH" is a symlink to the absolute pathname of the executing program.

"HAS_PSEUDOFORK"

This symbol, if defined, indicates that an emulation of the fork routine is available.

"HAS_REGCOMP"

This symbol, if defined, indicates that the "regcomp()" routine is available to do some regular pattern matching (usually on "POSIX".2 conforming systems).

"HAS_SETPGID"

This symbol, if defined, indicates that the "setpgid(pid, gpid)" routine is available to set process group ID.

"HAS_SIGSETJMP"

This variable indicates to the C program that the "sigsetjmp()" routine is available to save the calling process's registers and stack environment for later use by "siglongjmp()", and to optionally save the process's signal mask. See "Sigjmp_buf", "Sigsetjmp", and "Siglongjmp".

"HAS_STRUCT_CMSGHDR"

This symbol, if defined, indicates that the "struct cmsghdr" is supported.

"HAS_STRUCT_MSGHDR"

This symbol, if defined, indicates that the "struct msghdr" is supported.

"HAS_TGAMMA"

This symbol, if defined, indicates that the "tgamma" routine is available to do the gamma function. See also "HAS_LGAMMA".

"HAS_UNAME"

This symbol, if defined, indicates that the C program may use the "uname()" routine to derive the host name. See also "HAS_GETHOSTNAME" and "PHOSTNAME".

"HAS_UNION_SEMUN"

This symbol, if defined, indicates that the "union semun" is defined by including sys/sem.h. If not, the user code probably needs to define it as:

```
union semun {
int val;
struct semid_ds *buf;
```

```
 unsigned short *array;
 }
```

"I_DIRENT"

This symbol, if defined, indicates to the C program that it should include dirent.h.

Using this symbol also triggers the definition of the "Direntry_t" define which ends up being "'struct dirent'" or "'struct direct'" depending on the availability of dirent.h.

```
 #ifdef I_DIRENT
     #include <dirent.h>
 #endif
```

"I_POLL"

This symbol, if defined, indicates that poll.h exists and should be included. (see also "HAS_POLL")

```
 #ifdef I_POLL
     #include <poll.h>
 #endif
```

"I_SYS_RESOURCE"

This symbol, if defined, indicates to the C program that it should include sys/resource.h.

```
 #ifdef I_SYS_RESOURCE
     #include <sys_resource.h>
 #endif
```

"LIBM_LIB_VERSION"

This symbol, if defined, indicates that libm exports "_LIB_VERSION" and that math.h defines the enum to manipulate it.

"NEED_VA_COPY"

This symbol, if defined, indicates that the system stores the variable argument list datatype, "va_list", in a format that cannot be copied by simple assignment, so that some other means must be used when copying is required. As such systems vary in their provision (or non-provision) of copying mechanisms, handy.h defines a platform-independent macro, "Perl_va_copy(src, dst)", to do the job.

"OSNAME"

This symbol contains the name of the operating system, as determined by Configure.

You shouldn't rely on it too much; the specific feature tests from Configure are generally more reliable.

"OSVERS"

This symbol contains the version of the operating system, as determined by Configure. You shouldn't rely on it too much; the specific feature tests from Configure are generally more reliable.

"PHOSTNAME"

This symbol, if defined, indicates the command to feed to the "popen()" routine to derive the host name. See also "HAS_GETHOSTNAME" and "HAS_UNAME". Note that the command uses a fully qualified path, so that it is safe even if used by a process with super-user privileges.

"PROCSELFEXE_PATH"

If "HAS_PROCSELFEXE" is defined this symbol is the filename of the symbolic link pointing to the absolute pathname of the executing program.

"PTRSIZE"

This symbol contains the size of a pointer, so that the C preprocessor can make decisions based on it. It will be "sizeof(void *)" if the compiler supports (void *); otherwise it will be "sizeof(char *)".

"RANDBITS"

This symbol indicates how many bits are produced by the function used to generate normalized random numbers. Values include 15, 16, 31, and 48.

"SELECT_MIN_BITS"

This symbol holds the minimum number of bits operated by select. That is, if you do "select(n, ...)", how many bits at least will be cleared in the masks if some activity is detected. Usually this is either n or 32*"ceil(n/32)", especially many little-endians do the latter. This is only useful if you have "select()", naturally.

"SETUID_SCRIPTS_ARE_SECURE_NOW"

This symbol, if defined, indicates that the bug that prevents setuid scripts from being secure is not present in this kernel.

List of capability "HAS_foo" symbols

This is a list of those symbols that dont appear elsewhere in ths document that indicate if the current platform has a certain capability. Their names all begin with "HAS_". Only those symbols whose capability is directly derived from the name are listed here.

All others have their meaning expanded out elsewhere in this document. This (relatively) compact list is because we think that the expansion would add little or no value and take up a lot of space (because there are so many). If you think certain ones should be expanded, send email to perl5-porters@perl.org <mailto:perl5-porters@perl.org>. Each symbol here will be "#define"d if and only if the platform has the capability. If you need more detail, see the corresponding entry in config.h. For convenience, the list is split so that the ones that indicate there is a reentrant version of a capability are listed separately

"HAS_ACCEPT4",? "HAS_ACCESS",? "HAS_ACCESSX",? "HAS_ACOSH",? "HAS_AINTL",? "HAS_ALARM",? "HAS_ASINH",? "HAS_ATANH",? "HAS_ATOLL",? "HAS_CBRT",? "HAS_CHOWN",? "HAS_CHROOT",? "HAS_CHSIZE",? "HAS_CLEARENV",? "HAS_COPYSIGN",? "HAS_COPYSIGNL",? "HAS_CRYPT",? "HAS_CTERMID",? "HAS_CUSERID",? "HAS_DIRFD",? "HAS_DLADDR",? "HAS_DLERROR",? "HAS_EACCESS",? "HAS_ENDHOSTENT",? "HAS_ENDNETENT",? "HAS_ENDPROTOENT",? "HAS_ENDSERVENT",? "HAS_ERF",? "HAS_ERFC",? "HAS_EXP2",? "HAS_EXPM1",? "HAS_FCHMOD",? "HAS_FCHMODAT",? "HAS_FCHOWN",? "HAS_FDIM",? "HAS_FD_SET",? "HAS_FEGETROUND",? "HAS_FGETPOS",? "HAS_FLOCK",? "HAS_FMA",? "HAS_FMAX",? "HAS_FMIN",? "HAS_FORK",? "HAS_FSEEKO",? "HAS_FSETPOS",? "HAS_FSYNC",? "HAS_FTELLO",? "HAS_GAI_STRERROR",? "HAS_GETADDRINFO",? "HAS_GETCWD",? "HAS_GETESPWNAM",? "HAS_GETGROUPS",? "HAS_GETHOSTBYADDR",? "HAS_GETHOSTBYNAME",? "HAS_GETHOSTENT",? "HAS_GETLOGIN",? "HAS_GETNAMEINFO",? "HAS_GETNETBYADDR",? "HAS_GETNETBYNAME",? "HAS_GETNETENT",? "HAS_GETPAGESIZE",? "HAS_GETPGID",? "HAS_GETPGRP",? "HAS_GETPGRP2",? "HAS_GETPPID",? "HAS_GETPRIORITY",? "HAS_GETPROTOBYNAME",? "HAS_GETPROTOBYNUMBER",? "HAS_GETPROTOENT",? "HAS_GETPRPWNAM",? "HAS_GETSERVBYNAME",? "HAS_GETSERVBYPORT",? "HAS_GETSERVENT",? "HAS_GETSPNAM",? "HAS_HTONL",? "HAS_HTONS",? "HAS_HYPOT",? "HAS_ILOGBL",? "HAS_INETNTOP",? "HAS_INETPTON",? "HAS_INET_ATON",? "HAS_IPV6_MREQ",? "HAS_IPV6_MREQ_SOURCE",? "HAS_IP_MREQ",? "HAS_IP_MREQ_SOURCE",? "HAS_ISASCII",? "HAS_ISBLANK",? "HAS_ISLESS",? "HAS_KILLPG",? "HAS_LCHOWN",? "HAS_LINK",? "HAS_LINKAT",? "HAS_LLROUND",? "HAS_LOCKF",? "HAS_LOG1P",? "HAS_LOG2",? "HAS_LOGB",? "HAS_LROUND",? "HAS_LSTAT",? "HAS_MADVISE",? "HAS_MBLEN",? "HAS_MBRLEN",? "HAS_MBRTOWC",? "HAS_MBSTOWCS",? "HAS_MBTOWC",? "HAS_MEMMEM",? "HAS_MEMRCHR",? "HAS_MKDTEMP",? "HAS_MKFIFO",? "HAS_MKOSTEMP",? "HAS_MKSTEMP",? "HAS_MKSTEMPS",? "HAS_MMAP",? "HAS_MPROTECT",? "HAS_MSG",? "HAS_MSYNC",? "HAS_MUNMAP",? "HAS_NEARBYINT",? "HAS_NEXTAFTER",? "HAS_NICE",? "HAS_NTOHL",? "HAS_NTOHS",? "HAS_PATHCONF",? "HAS_PAUSE",? "HAS_PHOSTNAME",? "HAS_PIPE",? "HAS_PIPE2",?

"HAS_PRCTL",? "HAS_PTRDIFF_T",? "HAS_READLINK",? "HAS_READV",? "HAS_RECVMSG",?

"HAS_REMQUO",? "HAS_RENAME",? "HAS_RENAMEAT",? "HAS_RINT",? "HAS_ROUND",? "HAS_SCALBNL",?

"HAS_SEM",? "HAS_SENDMSG",? "HAS_SETEGID",? "HAS_SETEUID",? "HAS_SETGROUPS",?

"HAS_SETHOSTENT",? "HAS_SETLINEBUF",? "HAS_SETNETENT",? "HAS_SETPGRP",? "HAS_SETPGRP2",?

"HAS_SETPRIORITY",? "HAS_SETPROCTITLE",? "HAS_SETPROTOENT",? "HAS_SETREGID",?

"HAS_SETRESGID",? "HAS_SETRESUID",? "HAS_SETREUID",? "HAS_SETRGID",? "HAS_SETRUID",?

"HAS_SETSERVENT",? "HAS_SETSID",? "HAS_SHM",? "HAS_SIGACTION",? "HAS_SIGPROCMASK",?

"HAS_SIN6_SCOPE_ID",? "HAS_SNPRINTF",? "HAS_STAT",? "HAS_STRCOLL",? "HAS_STRERROR_L",?

"HAS_STRLCAT",? "HAS_STRLCPY",? "HAS_STRNLEN",? "HAS_STRTOD",? "HAS_STRTOL",?

"HAS_STRTOLL",? "HAS_STRTOQ",? "HAS_STRTOUL",? "HAS_STRTOULL",? "HAS_STRTOUQ",?

"HAS_STRXFRM",? "HAS_SYMLINK",? "HAS_SYSCALL",? "HAS_SYSCONF",? "HAS_SYSTEM",?

"HAS_SYS_ERRLIST",? "HAS_TCGETPGRP",? "HAS_TCSETPGRP",? "HAS_TOWLOWER",? "HAS_TOWUPPER",?

"HAS_TRUNCATE",? "HAS_TRUNCL",? "HAS_UALARM",? "HAS_UMASK",? "HAS_UNLINKAT",?

"HAS_UNSETENV",? "HAS_VFORK",? "HAS_VSNPRINTF",? "HAS_WAIT4",? "HAS_WAITPID",?

"HAS_WCRTOMB",? "HAS_WCSCMP",? "HAS_WCSTOMBS",? "HAS_WCSXFRM",? "HAS_WCTOMB",?

"HAS_WRITEV",? "HAS__FWALK"

And, the reentrant capabilities:

"HAS_CRYPT_R",? "HAS_CTERMID_R",? "HAS_DRAND48_R",? "HAS_ENDHOSTENT_R",?

"HAS_ENDNETENT_R",? "HAS_ENDPROTOENT_R",? "HAS_ENDSERVENT_R",? "HAS_GETGRGID_R",?

"HAS_GETGRNAM_R",? "HAS_GETHOSTBYADDR_R",? "HAS_GETHOSTBYNAME_R",?

"HAS_GETHOSTENT_R",?

"HAS_GETLOGIN_R",? "HAS_GETNETBYADDR_R",? "HAS_GETNETBYNAME_R",? "HAS_GETNETENT_R",?

"HAS_GETPROTOBYNAME_R",? "HAS_GETPROTOBYNUMBER_R",? "HAS_GETPROTOENT_R",?

"HAS_GETPWNAM_R",? "HAS_GETPWUID_R",? "HAS_GETSERVBYNAME_R",? "HAS_GETSERVBYPORT_R",?

"HAS_GETSERVENT_R",? "HAS_GETSPNAM_R",? "HAS_RANDOM_R",? "HAS_READDIR_R",?

"HAS_SETHOSTENT_R",? "HAS_SETNETENT_R",? "HAS_SETPROTOENT_R",? "HAS_SETSERVENT_R",?

"HAS_SRAND48_R",? "HAS_SRANDOM_R",? "HAS_STRERROR_R",? "HAS_TMPNAM_R",? "HAS_TTYNAME_R"

Example usage:

```
#ifdef HAS_STRNLEN
  use strnlen()
#else
  use an alternative implementation
#endif
```

List of "#include" needed symbols

This list contains symbols that indicate if certain "#include" files are present on the platform. If your code accesses the functionality that one of these is for, you will need to "#include" it if the symbol on this list is "#define"d. For more detail, see the corresponding entry in config.h.

"I_ARPA_INET",? "I_BFD",? "I_CRYPT",? "I_DBM",? "I_DLFCN",? "I_EXECINFO",? "I_FP",? "I_FP_CLASS",? "I_GDBM",? "I_GDBMNDBM",? "I_GDBM_NDBM",? "I_GRP",? "I_IEEEFP",? "I_INTTYPES",? "I_LIBUTIL",? "I_MNTENT",? "I_NDBM",? "I_NETDB",? "I_NETINET_IN",? "I_NETINET_TCP",? "I_NET_ERRNO",? "I_PROT",? "I_PWD",? "I_RPCSVC_DBM",? "I_SGTTY",? "I_SHADOW",? "I_STDBOOL",? "I_STDINT",? "I_SUNMATH",? "I_SYSLOG",? "I_SYSMODE",? "I_SYSUIO",? "I_SYSUTSNAME",? "I_SYS_ACCESS",? "I_SYS_IOCTL",? "I_SYS_MOUNT",? "I_SYS_PARAM",? "I_SYS_POLL",? "I_SYS_SECURITY",? "I_SYS_SELECT",? "I_SYS_STAT",? "I_SYS_STATVFS",? "I_SYS_TIME",? "I_SYS_TIMES",? "I_SYS_TIME_KERNEL",? "I_SYS_TYPES",? "I_SYS_UN",? "I_SYS_VFS",? "I_SYS_WAIT",? "I_TERMIO",? "I_TERMIOS",? "I_UNISTD",? "I_USTAT",? "I_VFORK",? "I_WCHAR",? "I_WCTYPE"

Example usage:

```
 #ifdef I_WCHAR
   #include <wchar.h>
 #endif
```

Global Variables

These variables are global to an entire process. They are shared between all interpreters and all threads in a process. Any variables not documented here may be changed or removed without notice, so don't use them! If you feel you really do need to use an unlisted variable, first send email to perl5-porters@perl.org <mailto:perl5-porters@perl.org>. It may be that someone there will point out a way to accomplish what you need without using an internal variable. But if not, you should get a go-ahead to document and then use the variable.

"PL_check"

Array, indexed by opcode, of functions that will be called for the "check" phase of optree building during compilation of Perl code. For most (but not all) types of op, once the op has been initially built and populated with child ops it will be filtered through the check function referenced by the appropriate element of this array. The new op is passed in as the sole argument to the check function, and the check function

returns the completed op. The check function may (as the name suggests) check the op for validity and signal errors. It may also initialise or modify parts of the ops, or perform more radical surgery such as adding or removing child ops, or even throw the op away and return a different op in its place.

This array of function pointers is a convenient place to hook into the compilation process. An XS module can put its own custom check function in place of any of the standard ones, to influence the compilation of a particular type of op. However, a custom check function must never fully replace a standard check function (or even a custom check function from another module). A module modifying checking must instead wrap the preexisting check function. A custom check function must be selective about when to apply its custom behaviour. In the usual case where it decides not to do anything special with an op, it must chain the preexisting op function. Check functions are thus linked in a chain, with the core's base checker at the end.

For thread safety, modules should not write directly to this array. Instead, use the function "wrap_op_checker".

"PL_keyword_plugin"

NOTE: "PL_keyword_plugin" is experimental and may change or be removed without notice.

Function pointer, pointing at a function used to handle extended keywords. The function should be declared as

```
int keyword_plugin_function(pTHX_
    char *keyword_ptr, STRLEN keyword_len,
    OP **op_ptr)
```

The function is called from the tokeniser, whenever a possible keyword is seen. "keyword_ptr" points at the word in the parser's input buffer, and "keyword_len" gives its length; it is not null-terminated. The function is expected to examine the word, and possibly other state such as %^H, to decide whether it wants to handle it as an extended keyword. If it does not, the function should return "KEYWORD_PLUGIN_DECLINE", and the normal parser process will continue.

If the function wants to handle the keyword, it first must parse anything following the keyword that is part of the syntax introduced by the keyword. See "Lexer interface" for details.

When a keyword is being handled, the plugin function must build a tree of "OP" structures, representing the code that was parsed. The root of the tree must be

stored in *op_ptr. The function then returns a constant indicating the syntactic role of the construct that it has parsed: "KEYWORD_PLUGIN_STMT" if it is a complete statement, or "KEYWORD_PLUGIN_EXPR" if it is an expression. Note that a statement construct cannot be used inside an expression (except via "do BLOCK" and similar), and an expression is not a complete statement (it requires at least a terminating semicolon).

When a keyword is handled, the plugin function may also have (compile-time) side effects. It may modify "%^H", define functions, and so on. Typically, if side effects are the main purpose of a handler, it does not wish to generate any ops to be included in the normal compilation. In this case it is still required to supply an op tree, but it suffices to generate a single null op.

That's how the *PL_keyword_plugin function needs to behave overall. Conventionally, however, one does not completely replace the existing handler function. Instead, take a copy of "PL_keyword_plugin" before assigning your own function pointer to it. Your handler function should look for keywords that it is interested in and handle those. Where it is not interested, it should call the saved plugin function, passing on the arguments it received. Thus "PL_keyword_plugin" actually points at a chain of handler functions, all of which have an opportunity to handle keywords, and only the last function in the chain (built into the Perl core) will normally return "KEYWORD_PLUGIN_DECLINE".

For thread safety, modules should not set this variable directly. Instead, use the function "wrap_keyword_plugin".

"PL_phase"

A value that indicates the current Perl interpreter's phase. Possible values include "PERL_PHASE_CONSTRUCT", "PERL_PHASE_START", "PERL_PHASE_CHECK", "PERL_PHASE_INIT", "PERL_PHASE_RUN", "PERL_PHASE_END", and "PERL_PHASE_DESTRUCT".

For example, the following determines whether the interpreter is in global destruction:

```
if (PL_phase == PERL_PHASE_DESTRUCT) {
    // we are in global destruction
}
```

"PL_phase" was introduced in Perl 5.14; in prior perls you can use "PL_dirty" (boolean) to determine whether the interpreter is in global destruction. (Use of

"PL_dirty" is discouraged since 5.14.)

```
enum perl_phase  PL_phase
```

GV Handling

A GV is a structure which corresponds to to a Perl typeglob, ie *foo. It is a structure that holds a pointer to a scalar, an array, a hash etc, corresponding to $foo, @foo, %foo. GVs are usually found as values in stashes (symbol table hashes) where Perl stores its global variables.

"gv_autoload4"

Equivalent to "gv_autoload_pvn".

```
GV*  gv_autoload4(HV* stash, const char* name, STRLEN len,
                  I32 method)
```

"GvAV"

Return the AV from the GV.

```
AV*  GvAV(GV* gv)
```

"gv_const_sv"

If "gv" is a typeglob whose subroutine entry is a constant sub eligible for inlining, or "gv" is a placeholder reference that would be promoted to such a typeglob, then returns the value returned by the sub. Otherwise, returns "NULL".

```
SV*  gv_const_sv(GV* gv)
```

"GvCV"

Return the CV from the GV.

```
CV*  GvCV(GV* gv)
```

"gv_fetchfile"

"gv_fetchfile_flags"

These return the debugger glob for the file (compiled by Perl) whose name is given by the "name" parameter.

There are currently exactly two differences between these functions.

The "name" parameter to "gv_fetchfile" is a C string, meaning it is "NUL"-terminated; whereas the "name" parameter to "gv_fetchfile_flags" is a Perl string, whose length (in bytes) is passed in via the "namelen" parameter This means the name may contain embedded "NUL" characters. "namelen" doesn't exist in plain "gv_fetchfile").

The other difference is that "gv_fetchfile_flags" has an extra "flags" parameter, which is currently completely ignored, but allows for possible future extensions.

GV* gv_fetchfile   (const char* name)

GV* gv_fetchfile_flags(const char *const name, const STRLEN len,

const U32 flags)

"gv_fetchmeth"

Like "gv_fetchmeth_pvn", but lacks a flags parameter.

GV* gv_fetchmeth(HV* stash, const char* name, STRLEN len,

I32 level)

"gv_fetchmethod"

See "gv_fetchmethod_autoload".

GV* gv_fetchmethod(HV* stash, const char* name)

"gv_fetchmethod_autoload"

Returns the glob which contains the subroutine to call to invoke the method on the "stash". In fact in the presence of autoloading this may be the glob for "AUTOLOAD". In this case the corresponding variable $AUTOLOAD is already setup.

The third parameter of "gv_fetchmethod_autoload" determines whether AUTOLOAD lookup is performed if the given method is not present: non-zero means yes, look for AUTOLOAD; zero means no, don't look for AUTOLOAD. Calling "gv_fetchmethod" is equivalent to calling "gv_fetchmethod_autoload" with a non-zero "autoload" parameter.

These functions grant "SUPER" token as a prefix of the method name. Note that if you want to keep the returned glob for a long time, you need to check for it being "AUTOLOAD", since at the later time the call may load a different subroutine due to $AUTOLOAD changing its value. Use the glob created as a side effect to do this.

These functions have the same side-effects as "gv_fetchmeth" with "level==0". The warning against passing the GV returned by "gv_fetchmeth" to "call_sv" applies equally to these functions.

GV* gv_fetchmethod_autoload(HV* stash, const char* name,

I32 autoload)

"gv_fetchmeth_autoload"

This is the old form of "gv_fetchmeth_pvn_autoload", which has no flags parameter.

GV* gv_fetchmeth_autoload(HV* stash, const char* name,

STRLEN len, I32 level)

"gv_fetchmeth_pv"

Exactly like "gv_fetchmeth_pvn", but takes a nul-terminated string instead of a

string/length pair.

```
 GV*  gv_fetchmeth_pv(HV* stash, const char* name, I32 level,
                      U32 flags)
```

"gv_fetchmeth_pvn"

Returns the glob with the given "name" and a defined subroutine or "NULL". The glob lives in the given "stash", or in the stashes accessible via @ISA and "UNIVERSAL::".

The argument "level" should be either 0 or -1. If "level==0", as a side-effect creates a glob with the given "name" in the given "stash" which in the case of success contains an alias for the subroutine, and sets up caching info for this glob.

The only significant values for "flags" are "GV_SUPER", "GV_NOUNIVERSAL", and "SVf_UTF8".

"GV_SUPER" indicates that we want to look up the method in the superclasses of the "stash".

"GV_NOUNIVERSAL" indicates that we do not want to look up the method in the stash accessible by "UNIVERSAL::".

The GV returned from "gv_fetchmeth" may be a method cache entry, which is not visible to Perl code. So when calling "call_sv", you should not use the GV directly; instead, you should use the method's CV, which can be obtained from the GV with the "GvCV" macro.

```
 GV*  gv_fetchmeth_pvn(HV* stash, const char* name, STRLEN len,
                       I32 level, U32 flags)
```

"gv_fetchmeth_pvn_autoload"

Same as "gv_fetchmeth_pvn()", but looks for autoloaded subroutines too. Returns a glob for the subroutine.

For an autoloaded subroutine without a GV, will create a GV even if "level < 0". For an autoloaded subroutine without a stub, "GvCV()" of the result may be zero.

Currently, the only significant value for "flags" is "SVf_UTF8".

```
 GV*  gv_fetchmeth_pvn_autoload(HV* stash, const char* name,
                                STRLEN len, I32 level, U32 flags)
```

"gv_fetchmeth_pv_autoload"

Exactly like "gv_fetchmeth_pvn_autoload", but takes a nul-terminated string instead of a string/length pair.

```
 GV*  gv_fetchmeth_pv_autoload(HV* stash, const char* name,
```

I32 level, U32 flags)

"gv_fetchmeth_sv"

Exactly like "gv_fetchmeth_pvn", but takes the name string in the form of an SV instead of a string/length pair.

GV* gv_fetchmeth_sv(HV* stash, SV* namesv, I32 level, U32 flags)

"gv_fetchmeth_sv_autoload"

Exactly like "gv_fetchmeth_pvn_autoload", but takes the name string in the form of an SV instead of a string/length pair.

GV* gv_fetchmeth_sv_autoload(HV* stash, SV* namesv, I32 level,
U32 flags)

"gv_fetchpv"

"gv_fetchpvn"

"gv_fetchpvn_flags"

"gv_fetchpvs"

"gv_fetchsv"

"gv_fetchsv_nomg"

These all return the GV of type "sv_type" whose name is given by the inputs, or NULL if no GV of that name and type could be found. See "Stashes and Globs" in perlguts.

The only differences are how the input name is specified, and if 'get' magic is normally used in getting that name.

Don't be fooled by the fact that only one form has "flags" in its name. They all have a "flags" parameter in fact, and all the flag bits have the same meanings for all

If any of the flags "GV_ADD", "GV_ADDMG", "GV_ADDWARN", "GV_ADDMULTI", or "GV_NOINIT" is set, a GV is created if none already exists for the input name and type. However, "GV_ADDMG" will only do the creation for magical GV's. For all of these flags except "GV_NOINIT", "gv_init_pvn" is called after the addition. "GV_ADDWARN" is used when the caller expects that adding won't be necessary because the symbol should already exist; but if not, add it anyway, with a warning that it was unexpectedly absent. The "GV_ADDMULTI" flag means to pretend that the GV has been seen before (i.e., suppress "Used once" warnings).

The flag "GV_NOADD_NOINIT" causes "gv_init_pvn" not be to called if the GV existed but isn't PVGV.

If the "SVf_UTF8" bit is set, the name is treated as being encoded in UTF-8; otherwise

the name won't be considered to be UTF-8 in the "pv"-named forms, and the UTF-8ness of the underlying SVs will be used in the "sv" forms.

If the flag "GV_NOTQUAL" is set, the caller warrants that the input name is a plain symbol name, not qualified with a package, otherwise the name is checked for being a qualified one.

In "gv_fetchpv", "nambeg" is a C string, NUL-terminated with no intermediate NULs.

In "gv_fetchpvs", "name" is a literal C string, hence is enclosed in double quotes.

"gv_fetchpvn" and "gv_fetchpvn_flags" are identical. In these, <nambeg> is a Perl string whose byte length is given by "full_len", and may contain embedded NULs.

In "gv_fetchsv" and "gv_fetchsv_nomg", the name is extracted from the PV of the input "name" SV. The only difference between these two forms is that 'get' magic is normally done on "name" in "gv_fetchsv", and always skipped with "gv_fetchsv_nomg". Including "GV_NO_SVGMAGIC" in the "flags" parameter to "gv_fetchsv" makes it behave identically to "gv_fetchsv_nomg".

```
GV*   gv_fetchpv       (const char *nambeg, I32 flags,
                        const svtype sv_type)
GV *  gv_fetchpvn      (const char * nambeg, STRLEN full_len,
                        I32 flags, const svtype sv_type)
GV*   gv_fetchpvn_flags(const char* name, STRLEN len, I32 flags,
                        const svtype sv_type)
GV *  gv_fetchpvs      ("name", I32 flags, const svtype sv_type)
GV*   gv_fetchsv       (SV *name, I32 flags, const svtype sv_type)
GV *  gv_fetchsv_nomg  (SV *name, I32 flags, const svtype sv_type)
```

"GvHV"

Return the HV from the GV.

```
HV*  GvHV(GV* gv)
```

"gv_init"

The old form of "gv_init_pvn()". It does not work with UTF-8 strings, as it has no flags parameter. If the "multi" parameter is set, the "GV_ADDMULTI" flag will be passed to "gv_init_pvn()".

```
void  gv_init(GV* gv, HV* stash, const char* name, STRLEN len,
              int multi)
```

"gv_init_pv"

Same as "gv_init_pvn()", but takes a nul-terminated string for the name instead of separate char * and length parameters.

```
 void  gv_init_pv(GV* gv, HV* stash, const char* name, U32 flags)
```

"gv_init_pvn"

Converts a scalar into a typeglob. This is an incoercible typeglob; assigning a reference to it will assign to one of its slots, instead of overwriting it as happens with typeglobs created by "SvSetSV". Converting any scalar that is "SvOK()" may produce unpredictable results and is reserved for perl's internal use.

"gv" is the scalar to be converted.

"stash" is the parent stash/package, if any.

"name" and "len" give the name. The name must be unqualified; that is, it must not include the package name. If "gv" is a stash element, it is the caller's responsibility to ensure that the name passed to this function matches the name of the element. If it does not match, perl's internal bookkeeping will get out of sync.

"flags" can be set to "SVf_UTF8" if "name" is a UTF-8 string, or the return value of SvUTF8(sv). It can also take the "GV_ADDMULTI" flag, which means to pretend that the GV has been seen before (i.e., suppress "Used once" warnings).

```
 void  gv_init_pvn(GV* gv, HV* stash, const char* name, STRLEN len,
                   U32 flags)
```

"gv_init_sv"

Same as "gv_init_pvn()", but takes an SV * for the name instead of separate char * and length parameters. "flags" is currently unused.

```
 void  gv_init_sv(GV* gv, HV* stash, SV* namesv, U32 flags)
```

"gv_stashpv"

Returns a pointer to the stash for a specified package. Uses "strlen" to determine the length of "name", then calls "gv_stashpvn()".

```
 HV*  gv_stashpv(const char* name, I32 flags)
```

"gv_stashpvn"

Returns a pointer to the stash for a specified package. The "namelen" parameter indicates the length of the "name", in bytes. "flags" is passed to "gv_fetchpvn_flags()", so if set to "GV_ADD" then the package will be created if it does not already exist. If the package does not exist and "flags" is 0 (or any other setting that does not create packages) then "NULL" is returned.

Flags may be one of:

GV_ADD Create and initialize the package if doesn't already exist

GV_NOADD_NOINIT Don't create the package,

GV_ADDMG GV_ADD iff the GV is magical

GV_NOINIT GV_ADD, but don't initialize

GV_NOEXPAND Don't expand SvOK() entries to PVGV

SVf_UTF8 The name is in UTF-8

The most important of which are probably "GV_ADD" and "SVf_UTF8".

Note, use of "gv_stashsv" instead of "gv_stashpvn" where possible is strongly recommended for performance reasons.

HV* gv_stashpvn(const char* name, U32 namelen, I32 flags)

"gv_stashpvs"

Like "gv_stashpvn", but takes a literal string instead of a string/length pair.

HV* gv_stashpvs("name", I32 create)

"gv_stashsv"

Returns a pointer to the stash for a specified package. See "gv_stashpvn".

Note this interface is strongly preferred over "gv_stashpvn" for performance reasons.

HV* gv_stashsv(SV* sv, I32 flags)

"GvSV"

Return the SV from the GV.

Prior to Perl v5.9.3, this would add a scalar if none existed. Nowadays, use "GvSVn" for that, or compile perl with "-DPERL_CREATE_GVSV". See perl5100delta.

SV* GvSV(GV* gv)

"GvSVn"

Like "GvSV", but creates an empty scalar if none already exists.

SV* GvSVn(GV* gv)

"save_gp"

Saves the current GP of gv on the save stack to be restored on scope exit.

If empty is true, replace the GP with a new GP.

If empty is false, mark gv with GVf_INTRO so the next reference assigned is localized, which is how " local *foo = $someref; " works.

void save_gp(GV* gv, I32 empty)

"setdefout"

Sets "PL_defoutgv", the default file handle for output, to the passed in typeglob. As "PL_defoutgv" "owns" a reference on its typeglob, the reference count of the passed in typeglob is increased by one, and the reference count of the typeglob that "PL_defoutgv" points to is decreased by one.

```
void  setdefout(GV* gv)
```

Hook manipulation

These functions provide convenient and thread-safe means of manipulating hook variables.

"wrap_op_checker"

Puts a C function into the chain of check functions for a specified op type. This is the preferred way to manipulate the "PL_check" array. "opcode" specifies which type of op is to be affected. "new_checker" is a pointer to the C function that is to be added to that opcode's check chain, and "old_checker_p" points to the storage location where a pointer to the next function in the chain will be stored. The value of "new_checker" is written into the "PL_check" array, while the value previously stored there is written to *old_checker_p.

"PL_check" is global to an entire process, and a module wishing to hook op checking may find itself invoked more than once per process, typically in different threads. To handle that situation, this function is idempotent. The location *old_checker_p must initially (once per process) contain a null pointer. A C variable of static duration (declared at file scope, typically also marked "static" to give it internal linkage) will be implicitly initialised appropriately, if it does not have an explicit initialiser. This function will only actually modify the check chain if it finds *old_checker_p to be null. This function is also thread safe on the small scale. It uses appropriate locking to avoid race conditions in accessing "PL_check".

When this function is called, the function referenced by "new_checker" must be ready to be called, except for *old_checker_p being unfilled. In a threading situation, "new_checker" may be called immediately, even before this function has returned. *old_checker_p will always be appropriately set before "new_checker" is called. If "new_checker" decides not to do anything special with an op that it is given (which is the usual case for most uses of op check hooking), it must chain the check function referenced by *old_checker_p.

Taken all together, XS code to hook an op checker should typically look something like

this:

```
static Perl_check_t nxck_frob;
static OP *myck_frob(pTHX_ OP *op) {
    ...
    op = nxck_frob(aTHX_ op);
    ...
    return op;
}
BOOT:
    wrap_op_checker(OP_FROB, myck_frob, &nxck_frob);
```

If you want to influence compilation of calls to a specific subroutine, then use "cv_set_call_checker_flags" rather than hooking checking of all "entersub" ops.

```
void  wrap_op_checker(Optype opcode, Perl_check_t new_checker,
                      Perl_check_t *old_checker_p)
```

HV Handling

A HV structure represents a Perl hash. It consists mainly of an array of pointers, each of which points to a linked list of HE structures. The array is indexed by the hash function of the key, so each linked list represents all the hash entries with the same hash value. Each HE contains a pointer to the actual value, plus a pointer to a HEK structure which holds the key and hash value.

"get_hv"

Returns the HV of the specified Perl hash. "flags" are passed to "gv_fetchpv". If "GV_ADD" is set and the Perl variable does not exist then it will be created. If "flags" is zero and the variable does not exist then "NULL" is returned.

NOTE: the "perl_get_hv()" form is deprecated.

```
HV*  get_hv(const char *name, I32 flags)
```

"HEf_SVKEY"

This flag, used in the length slot of hash entries and magic structures, specifies the structure contains an "SV*" pointer where a "char*" pointer is to be expected. (For information only--not to be used).

"HeHASH"

Returns the computed hash stored in the hash entry.

```
U32  HeHASH(HE* he)
```

"HeKEY"

Returns the actual pointer stored in the key slot of the hash entry. The pointer may be either "char*" or "SV*", depending on the value of "HeKLEN()". Can be assigned to. The "HePV()" or "HeSVKEY()" macros are usually preferable for finding the value of a key.

```
void*  HeKEY(HE* he)
```

"HeKLEN"

If this is negative, and amounts to "HEf_SVKEY", it indicates the entry holds an "SV*" key. Otherwise, holds the actual length of the key. Can be assigned to. The "HePV()" macro is usually preferable for finding key lengths.

```
STRLEN  HeKLEN(HE* he)
```

"HePV"

Returns the key slot of the hash entry as a "char*" value, doing any necessary dereferencing of possibly "SV*" keys. The length of the string is placed in "len" (this is a macro, so do not use &len). If you do not care about what the length of the key is, you may use the global variable "PL_na", though this is rather less efficient than using a local variable. Remember though, that hash keys in perl are free to contain embedded nulls, so using "strlen()" or similar is not a good way to find the length of hash keys. This is very similar to the "SvPV()" macro described elsewhere in this document. See also "HeUTF8".

If you are using "HePV" to get values to pass to "newSVpvn()" to create a new SV, you should consider using "newSVhek(HeKEY_hek(he))" as it is more efficient.

```
char*  HePV(HE* he, STRLEN len)
```

"HeSVKEY"

Returns the key as an "SV*", or "NULL" if the hash entry does not contain an "SV*" key.

```
SV*  HeSVKEY(HE* he)
```

"HeSVKEY_force"

Returns the key as an "SV*". Will create and return a temporary mortal "SV*" if the hash entry contains only a "char*" key.

```
SV*  HeSVKEY_force(HE* he)
```

"HeSVKEY_set"

Sets the key to a given "SV*", taking care to set the appropriate flags to indicate

the presence of an "SV*" key, and returns the same "SV*".

SV* HeSVKEY_set(HE* he, SV* sv)

"HeUTF8"

Returns whether the "char *" value returned by "HePV" is encoded in UTF-8, doing any necessary dereferencing of possibly "SV*" keys. The value returned will be 0 or non-0, not necessarily 1 (or even a value with any low bits set), so do not blindly assign this to a "bool" variable, as "bool" may be a typedef for "char".

U32 HeUTF8(HE* he)

"HeVAL"

Returns the value slot (type "SV*") stored in the hash entry. Can be assigned to.

SV *foo= HeVAL(hv);

HeVAL(hv)= sv;

SV* HeVAL(HE* he)

"HV"

Described in perlguts.

"hv_assert"

Check that a hash is in an internally consistent state.

NOTE: "hv_assert" must be explicitly called as "Perl_hv_assert" with an "aTHX_" parameter.

void Perl_hv_assert(pTHX_ HV *hv)

"hv_bucket_ratio"

NOTE: "hv_bucket_ratio" is experimental and may change or be removed without notice.

If the hash is tied dispatches through to the SCALAR tied method, otherwise if the hash contains no keys returns 0, otherwise returns a mortal sv containing a string specifying the number of used buckets, followed by a slash, followed by the number of available buckets.

This function is expensive, it must scan all of the buckets to determine which are used, and the count is NOT cached. In a large hash this could be a lot of buckets.

SV* hv_bucket_ratio(HV *hv)

"hv_clear"

Frees all the elements of a hash, leaving it empty. The XS equivalent of "%hash = ()". See also "hv_undef".

See "av_clear" for a note about the hash possibly being invalid on return.

void hv_clear(HV *hv)

"hv_clear_placeholders"

Clears any placeholders from a hash. If a restricted hash has any of its keys marked as readonly and the key is subsequently deleted, the key is not actually deleted but is marked by assigning it a value of &PL_sv_placeholder. This tags it so it will be ignored by future operations such as iterating over the hash, but will still allow the hash to have a value reassigned to the key at some future point. This function clears any such placeholder keys from the hash. See "Hash::Util::lock_keys()" for an example of its use.

void hv_clear_placeholders(HV *hv)

"hv_copy_hints_hv"

A specialised version of "newHVhv" for copying "%^H". "ohv" must be a pointer to a hash (which may have "%^H" magic, but should be generally non-magical), or "NULL" (interpreted as an empty hash). The content of "ohv" is copied to a new hash, which has the "%^H"-specific magic added to it. A pointer to the new hash is returned.

HV * hv_copy_hints_hv(HV *const ohv)

"hv_delete"

Deletes a key/value pair in the hash. The value's SV is removed from the hash, made mortal, and returned to the caller. The absolute value of "klen" is the length of the key. If "klen" is negative the key is assumed to be in UTF-8-encoded Unicode. The "flags" value will normally be zero; if set to "G_DISCARD" then "NULL" will be returned. "NULL" will also be returned if the key is not found.

SV* hv_delete(HV *hv, const char *key, I32 klen, I32 flags)

"hv_delete_ent"

Deletes a key/value pair in the hash. The value SV is removed from the hash, made mortal, and returned to the caller. The "flags" value will normally be zero; if set to "G_DISCARD" then "NULL" will be returned. "NULL" will also be returned if the key is not found. "hash" can be a valid precomputed hash value, or 0 to ask for it to be computed.

SV* hv_delete_ent(HV *hv, SV *keysv, I32 flags, U32 hash)

"HvENAME"

Returns the effective name of a stash, or NULL if there is none. The effective name represents a location in the symbol table where this stash resides. It is updated

automatically when packages are aliased or deleted. A stash that is no longer in the symbol table has no effective name. This name is preferable to "HvNAME" for use in MRO linearisations and isa caches.

```
char*  HvENAME(HV* stash)
```

"HvENAMELEN"

Returns the length of the stash's effective name.

```
STRLEN  HvENAMELEN(HV *stash)
```

"HvENAMEUTF8"

Returns true if the effective name is in UTF-8 encoding.

```
unsigned char  HvENAMEUTF8(HV *stash)
```

"hv_exists"

Returns a boolean indicating whether the specified hash key exists. The absolute value of "klen" is the length of the key. If "klen" is negative the key is assumed to be in UTF-8-encoded Unicode.

```
bool  hv_exists(HV *hv, const char *key, I32 klen)
```

"hv_exists_ent"

Returns a boolean indicating whether the specified hash key exists. "hash" can be a valid precomputed hash value, or 0 to ask for it to be computed.

```
bool  hv_exists_ent(HV *hv, SV *keysv, U32 hash)
```

"hv_fetch"

Returns the SV which corresponds to the specified key in the hash. The absolute value of "klen" is the length of the key. If "klen" is negative the key is assumed to be in UTF-8-encoded Unicode. If "lval" is set then the fetch will be part of a store. This means that if there is no value in the hash associated with the given key, then one is created and a pointer to it is returned. The "SV*" it points to can be assigned to. But always check that the return value is non-null before dereferencing it to an "SV*".

See "Understanding the Magic of Tied Hashes and Arrays" in perlguts for more information on how to use this function on tied hashes.

```
SV**  hv_fetch(HV *hv, const char *key, I32 klen, I32 lval)
```

"hv_fetchs"

Like "hv_fetch", but takes a literal string instead of a string/length pair.

```
SV**  hv_fetchs(HV* tb, "key", I32 lval)
```

"hv_fetch_ent"

Returns the hash entry which corresponds to the specified key in the hash. "hash" must be a valid precomputed hash number for the given "key", or 0 if you want the function to compute it. IF "lval" is set then the fetch will be part of a store. Make sure the return value is non-null before accessing it. The return value when "hv" is a tied hash is a pointer to a static location, so be sure to make a copy of the structure if you need to store it somewhere.

See "Understanding the Magic of Tied Hashes and Arrays" in perlguts for more information on how to use this function on tied hashes.

```
HE*  hv_fetch_ent(HV *hv, SV *keysv, I32 lval, U32 hash)
```

"HvFILL"

See "hv_fill".

```
STRLEN  HvFILL(HV *const hv)
```

"hv_fill"

Returns the number of hash buckets that happen to be in use.

This function is wrapped by the macro "HvFILL".

As of perl 5.25 this function is used only for debugging purposes, and the number of used hash buckets is not in any way cached, thus this function can be costly to execute as it must iterate over all the buckets in the hash.

NOTE: "hv_fill" must be explicitly called as "Perl_hv_fill" with an "aTHX_" parameter.

```
STRLEN  Perl_hv_fill(pTHX_ HV *const hv)
```

"hv_iterinit"

Prepares a starting point to traverse a hash table. Returns the number of keys in the hash, including placeholders (i.e. the same as "HvTOTALKEYS(hv)"). The return value is currently only meaningful for hashes without tie magic.

NOTE: Before version 5.004_65, "hv_iterinit" used to return the number of hash buckets that happen to be in use. If you still need that esoteric value, you can get it through the macro "HvFILL(hv)".

```
I32  hv_iterinit(HV *hv)
```

"hv_iterkey"

Returns the key from the current position of the hash iterator. See "hv_iterinit".

```
char*  hv_iterkey(HE* entry, I32* retlen)
```

"hv_iterkeysv"

Returns the key as an "SV*" from the current position of the hash iterator. The return value will always be a mortal copy of the key. Also see "hv_iterinit".

```
SV*  hv_iterkeysv(HE* entry)
```

"hv_iternext"

Returns entries from a hash iterator. See "hv_iterinit".

You may call "hv_delete" or "hv_delete_ent" on the hash entry that the iterator currently points to, without losing your place or invalidating your iterator. Note that in this case the current entry is deleted from the hash with your iterator holding the last reference to it. Your iterator is flagged to free the entry on the next call to "hv_iternext", so you must not discard your iterator immediately else the entry will leak - call "hv_iternext" to trigger the resource deallocation.

```
HE*  hv_iternext(HV *hv)
```

"hv_iternextsv"

Performs an "hv_iternext", "hv_iterkey", and "hv_iterval" in one operation.

```
SV*  hv_iternextsv(HV *hv, char **key, I32 *retlen)
```

"hv_iternext_flags"

NOTE: "hv_iternext_flags" is experimental and may change or be removed without notice.

Returns entries from a hash iterator. See "hv_iterinit" and "hv_iternext". The "flags" value will normally be zero; if "HV_ITERNEXT_WANTPLACEHOLDERS" is set the placeholders keys (for restricted hashes) will be returned in addition to normal keys. By default placeholders are automatically skipped over. Currently a placeholder is implemented with a value that is &PL_sv_placeholder. Note that the implementation of placeholders and restricted hashes may change, and the implementation currently is insufficiently abstracted for any change to be tidy.

```
HE*  hv_iternext_flags(HV *hv, I32 flags)
```

"hv_iterval"

Returns the value from the current position of the hash iterator. See "hv_iterkey".

```
SV*  hv_iterval(HV *hv, HE *entry)
```

"hv_magic"

Adds magic to a hash. See "sv_magic".

```
void  hv_magic(HV *hv, GV *gv, int how)
```

"HvNAME"

Returns the package name of a stash, or "NULL" if "stash" isn't a stash. See

"SvSTASH", "CvSTASH".

char* HvNAME(HV* stash)

"HvNAMELEN"

Returns the length of the stash's name.

Disfavored forms of HvNAME and HvNAMELEN; suppress mention of them

STRLEN HvNAMELEN(HV *stash)

"HvNAMEUTF8"

Returns true if the name is in UTF-8 encoding.

unsigned char HvNAMEUTF8(HV *stash)

"hv_scalar"

Evaluates the hash in scalar context and returns the result.

When the hash is tied dispatches through to the SCALAR method, otherwise returns a mortal SV containing the number of keys in the hash.

Note, prior to 5.25 this function returned what is now returned by the hv_bucket_ratio() function.

SV* hv_scalar(HV *hv)

"hv_store"

Stores an SV in a hash. The hash key is specified as "key" and the absolute value of "klen" is the length of the key. If "klen" is negative the key is assumed to be in UTF-8-encoded Unicode. The "hash" parameter is the precomputed hash value; if it is zero then Perl will compute it.

The return value will be "NULL" if the operation failed or if the value did not need to be actually stored within the hash (as in the case of tied hashes). Otherwise it can be dereferenced to get the original "SV*". Note that the caller is responsible for suitably incrementing the reference count of "val" before the call, and decrementing it if the function returned "NULL". Effectively a successful "hv_store" takes ownership of one reference to "val". This is usually what you want; a newly created SV has a reference count of one, so if all your code does is create SVs then store them in a hash, "hv_store" will own the only reference to the new SV, and your code doesn't need to do anything further to tidy up. "hv_store" is not implemented as a call to "hv_store_ent", and does not create a temporary SV for the key, so if your key data is not already in SV form then use "hv_store" in preference to "hv_store_ent".

See "Understanding the Magic of Tied Hashes and Arrays" in perlguts for more information on how to use this function on tied hashes.

```
SV**  hv_store(HV *hv, const char *key, I32 klen, SV *val,
               U32 hash)
```

"hv_stores"

Like "hv_store", but takes a literal string instead of a string/length pair and omits the hash parameter.

```
SV**  hv_stores(HV* tb, "key", SV* val)
```

"hv_store_ent"

Stores "val" in a hash. The hash key is specified as "key". The "hash" parameter is the precomputed hash value; if it is zero then Perl will compute it. The return value is the new hash entry so created. It will be "NULL" if the operation failed or if the value did not need to be actually stored within the hash (as in the case of tied hashes). Otherwise the contents of the return value can be accessed using the "He?" macros described here. Note that the caller is responsible for suitably incrementing the reference count of "val" before the call, and decrementing it if the function returned NULL. Effectively a successful "hv_store_ent" takes ownership of one reference to "val". This is usually what you want; a newly created SV has a reference count of one, so if all your code does is create SVs then store them in a hash, "hv_store" will own the only reference to the new SV, and your code doesn't need to do anything further to tidy up. Note that "hv_store_ent" only reads the "key"; unlike "val" it does not take ownership of it, so maintaining the correct reference count on "key" is entirely the caller's responsibility. The reason it does not take ownership, is that "key" is not used after this function returns, and so can be freed immediately. "hv_store" is not implemented as a call to "hv_store_ent", and does not create a temporary SV for the key, so if your key data is not already in SV form then use "hv_store" in preference to "hv_store_ent".

See "Understanding the Magic of Tied Hashes and Arrays" in perlguts for more information on how to use this function on tied hashes.

```
HE*  hv_store_ent(HV *hv, SV *key, SV *val, U32 hash)
```

"hv_undef"

Undefines the hash. The XS equivalent of "undef(%hash)".

As well as freeing all the elements of the hash (like "hv_clear()"), this also frees

any auxiliary data and storage associated with the hash.

See "av_clear" for a note about the hash possibly being invalid on return.

```
void  hv_undef(HV *hv)
```

"MGVTBL"

Described in perlguts.

"newHV"

Creates a new HV. The reference count is set to 1.

```
HV*  newHV()
```

"Nullhv"

"DEPRECATED!" It is planned to remove "Nullhv" from a future release of Perl. Do not use it for new code; remove it from existing code.

Null HV pointer.

(deprecated - use "(HV *)NULL" instead)

"PERL_HASH"

Described in perlguts.

```
void  PERL_HASH(U32 hash, char *key, STRLEN klen)
```

"PERL_MAGIC_arylen"

"PERL_MAGIC_arylen_p"

"PERL_MAGIC_backref"

"PERL_MAGIC_bm"

"PERL_MAGIC_checkcall"

"PERL_MAGIC_collxfrm"

"PERL_MAGIC_dbfile"

"PERL_MAGIC_dbline"

"PERL_MAGIC_debugvar"

"PERL_MAGIC_defelem"

"PERL_MAGIC_env"

"PERL_MAGIC_envelem"

"PERL_MAGIC_ext"

"PERL_MAGIC_fm"

"PERL_MAGIC_hints"

"PERL_MAGIC_hintselem"

"PERL_MAGIC_isa"

"PERL_MAGIC_isaelem"

"PERL_MAGIC_lvref"

"PERL_MAGIC_nkeys"

"PERL_MAGIC_nonelem"

"PERL_MAGIC_overload_table"

"PERL_MAGIC_pos"

"PERL_MAGIC_qr"

"PERL_MAGIC_regdata"

"PERL_MAGIC_regdatum"

"PERL_MAGIC_regex_global"

"PERL_MAGIC_rhash"

"PERL_MAGIC_shared"

"PERL_MAGIC_shared_scalar"

"PERL_MAGIC_sig"

"PERL_MAGIC_sigelem"

"PERL_MAGIC_substr"

"PERL_MAGIC_sv"

"PERL_MAGIC_symtab"

"PERL_MAGIC_taint"

"PERL_MAGIC_tied"

"PERL_MAGIC_tiedelem"

"PERL_MAGIC_tiedscalar"

"PERL_MAGIC_utf8"

"PERL_MAGIC_uvar"

"PERL_MAGIC_uvar_elem"

"PERL_MAGIC_vec"

"PERL_MAGIC_vstring"

Described in perlguts.

"PL_modglobal"

"PL_modglobal" is a general purpose, interpreter global HV for use by extensions that need to keep information on a per-interpreter basis. In a pinch, it can also be used as a symbol table for extensions to share data among each other. It is a good idea to use keys prefixed by the package name of the extension that owns the data.

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

HV* PL_modglobal

Input/Output

"PerlIO_apply_layers"

Described in perlapio.

int PerlIO_apply_layers(PerlIO *f, const char *mode,
const char *layers)

"PerlIO_binmode"

Described in perlapio.

int PerlIO_binmode(PerlIO *f, int ptype, int imode,
const char *layers)

"PerlIO_canset_cnt"

Described in perlapio.

int PerlIO_canset_cnt(PerlIO *f)

"PerlIO_clearerr"

Described in perlapio.

void PerlIO_clearerr(PerlIO *f)

"PerlIO_close"

Described in perlapio.

int PerlIO_close(PerlIO *f)

"PerlIO_debug"

Described in perlapio.

void PerlIO_debug(const char *fmt, ...)

"PerlIO_eof"

Described in perlapio.

int PerlIO_eof(PerlIO *f)

"PerlIO_error"

Described in perlapio.

int PerlIO_error(PerlIO *f)

"PerlIO_exportFILE"

Described in perlapio.

FILE * PerlIO_exportFILE(PerlIO *f, const char *mode)

"PerlIO_fast_gets"

Described in perlapio.

int PerlIO_fast_gets(PerlIO *f)

"PerlIO_fdopen"

Described in perlapio.

PerlIO* PerlIO_fdopen(int fd, const char *mode)

"PerlIO_fileno"

Described in perlapio.

int PerlIO_fileno(PerlIO *f)

"PerlIO_findFILE"

Described in perlapio.

FILE * PerlIO_findFILE(PerlIO *f)

"PerlIO_flush"

Described in perlapio.

int PerlIO_flush(PerlIO *f)

"PERLIO_F_APPEND"

"PERLIO_F_CANREAD"

"PERLIO_F_CANWRITE"

"PERLIO_F_CRLF"

"PERLIO_F_EOF"

"PERLIO_F_ERROR"

"PERLIO_F_FASTGETS"

"PERLIO_F_LINEBUF"

"PERLIO_F_OPEN"

"PERLIO_F_RDBUF"

"PERLIO_F_TEMP"

"PERLIO_F_TRUNCATE"

"PERLIO_F_UNBUF"

"PERLIO_F_UTF8"

"PERLIO_F_WRBUF"

Described in perliol.

"PerlIO_getc"

Described in perlapio.

int PerlIO_getc(PerlIO *d)

"PerlIO_getpos"

Described in perlapio.

int PerlIO_getpos(PerlIO *f, SV *save)

"PerlIO_get_base"

Described in perlapio.

STDCHAR * PerlIO_get_base(PerlIO *f)

"PerlIO_get_bufsiz"

Described in perlapio.

SSize_t PerlIO_get_bufsiz(PerlIO *f)

"PerlIO_get_cnt"

Described in perlapio.

SSize_t PerlIO_get_cnt(PerlIO *f)

"PerlIO_get_ptr"

Described in perlapio.

STDCHAR * PerlIO_get_ptr(PerlIO *f)

"PerlIO_has_base"

Described in perlapio.

int PerlIO_has_base(PerlIO *f)

"PerlIO_has_cntptr"

Described in perlapio.

int PerlIO_has_cntptr(PerlIO *f)

"PerlIO_importFILE"

Described in perlapio.

PerlIO* PerlIO_importFILE(FILE *stdio, const char *mode)

"PERLIO_K_BUFFERED"

"PERLIO_K_CANCRLF"

"PERLIO_K_FASTGETS"

"PERLIO_K_MULTIARG"

"PERLIO_K_RAW"

Described in perliol.

"PerlIO_open"

Described in perlapio.

PerlIO* PerlIO_open(const char *path, const char *mode)

"PerlIO_printf"

Described in perlapio.

int PerlIO_printf(PerlIO *f, const char *fmt, ...)

"PerlIO_putc"

Described in perlapio.

int PerlIO_putc(PerlIO *f, int ch)

"PerlIO_puts"

Described in perlapio.

int PerlIO_puts(PerlIO *f, const char *string)

"PerlIO_read"

Described in perlapio.

SSize_t PerlIO_read(PerlIO *f, void *vbuf, Size_t count)

"PerlIO_releaseFILE"

Described in perlapio.

void PerlIO_releaseFILE(PerlIO *f, FILE *stdio)

"PerlIO_reopen"

Described in perlapio.

PerlIO * PerlIO_reopen(const char *path, const char *mode,

PerlIO *old)

"PerlIO_rewind"

Described in perlapio.

void PerlIO_rewind(PerlIO *f)

"PerlIO_seek"

Described in perlapio.

int PerlIO_seek(PerlIO *f, Off_t offset, int whence)

"PerlIO_setlinebuf"

Described in perlapio.

void PerlIO_setlinebuf(PerlIO *f)

"PerlIO_setpos"

Described in perlapio.

int PerlIO_setpos(PerlIO *f, SV *saved)

"PerlIO_set_cnt"

Described in perlapio.

void PerlIO_set_cnt(PerlIO *f, SSize_t cnt)

"PerlIO_set_ptrcnt"

Described in perlapio.

void PerlIO_set_ptrcnt(PerlIO *f, STDCHAR *ptr, SSize_t cnt)

"PerlIO_stderr"

Described in perlapio.

PerlIO * PerlIO_stderr()

"PerlIO_stdin"

Described in perlapio.

PerlIO * PerlIO_stdin()

"PerlIO_stdout"

Described in perlapio.

PerlIO * PerlIO_stdout()

"PerlIO_stdoutf"

Described in perlapio.

int PerlIO_stdoutf(const char *fmt, ...)

"PerlIO_tell"

Described in perlapio.

Off_t PerlIO_tell(PerlIO *f)

"PerlIO_ungetc"

Described in perlapio.

int PerlIO_ungetc(PerlIO *f, int ch)

"PerlIO_vprintf"

Described in perlapio.

int PerlIO_vprintf(PerlIO *f, const char *fmt, va_list args)

"PerlIO_write"

Described in perlapio.

SSize_t PerlIO_write(PerlIO *f, const void *vbuf, Size_t count)

"PL_maxsysfd"

Described in perliol.

Integer configuration values

"CASTI32"

This symbol is defined if the C compiler can cast negative or large floating point numbers to 32-bit ints.

"HAS_INT64_T"

This symbol will defined if the C compiler supports "int64_t". Usually the inttypes.h needs to be included, but sometimes sys/types.h is enough.

"HAS_LONG_LONG"

This symbol will be defined if the C compiler supports long long.

"HAS_QUAD"

This symbol, if defined, tells that there's a 64-bit integer type, "Quad_t", and its unsigned counterpart, "Uquad_t". "QUADKIND" will be one of "QUAD_IS_INT", "QUAD_IS_LONG", "QUAD_IS_LONG_LONG", "QUAD_IS_INT64_T", or "QUAD_IS___INT64".

"HE"

Described in perlguts.

"I8"

"I16"

"I32"

"I64"

"IV"

Described in perlguts.

"I32SIZE"

This symbol contains the "sizeof(I32)".

"I32TYPE"

This symbol defines the C type used for Perl's I32.

"I64SIZE"

This symbol contains the "sizeof(I64)".

"I64TYPE"

This symbol defines the C type used for Perl's I64.

"I16SIZE"

This symbol contains the "sizeof(I16)".

"I16TYPE"

This symbol defines the C type used for Perl's I16.

"INT16_C"

"INT32_C"

"INT64_C"

Returns a token the C compiler recognizes for the constant "number" of the corresponding integer type on the machine.

If the machine does not have a 64-bit type, "INT64_C" is undefined. Use "INTMAX_C" to get the largest type available on the platform.

```
I16  INT16_C(number)
I32  INT32_C(number)
I64  INT64_C(number)
```

"INTMAX_C"

Returns a token the C compiler recognizes for the constant "number" of the widest integer type on the machine. For example, if the machine has "long long"s, "INTMAX_C(-1)" would yield

```
-1LL
```

See also, for example, "INT32_C".

Use "IV" to declare variables of the maximum usable size on this platform.

```
INTMAX_C(number)
```

"INTSIZE"

This symbol contains the value of "sizeof(int)" so that the C preprocessor can make decisions based on it.

"I8SIZE"

This symbol contains the "sizeof(I8)".

"I8TYPE"

This symbol defines the C type used for Perl's I8.

"IV_MAX"

The largest signed integer that fits in an IV on this platform.

```
IV  IV_MAX
```

"IV_MIN"

The negative signed integer furthest away from 0 that fits in an IV on this platform.

```
IV  IV_MIN
```

"IVSIZE"

This symbol contains the "sizeof(IV)".

"IVTYPE"

This symbol defines the C type used for Perl's IV.

"line_t"

The typedef to use to declare variables that are to hold line numbers.

"LONGLONGSIZE"

This symbol contains the size of a long long, so that the C preprocessor can make decisions based on it. It is only defined if the system supports long long.

"LONGSIZE"

This symbol contains the value of "sizeof(long)" so that the C preprocessor can make decisions based on it.

"memzero"

Set the "l" bytes starting at *d to all zeroes.

```
 void  memzero(void * d, Size_t l)
```

"NV"

Described in perlguts.

"PERL_INT_FAST8_T"

"PERL_INT_FAST16_T"

"PERL_UINT_FAST8_T"

"PERL_UINT_FAST16_T"

These are equivalent to the correspondingly-named C99 typedefs on platforms that have those; they evaluate to "int" and "unsigned int" on platforms that don't, so that you can portably take advantage of this C99 feature.

"PERL_INT_MAX"

"PERL_INT_MIN"

"PERL_LONG_MAX"

"PERL_LONG_MIN"

"PERL_SHORT_MAX"

"PERL_SHORT_MIN"

"PERL_UCHAR_MAX"

"PERL_UCHAR_MIN"

"PERL_UINT_MAX"

"PERL_UINT_MIN"

"PERL_ULONG_MAX"

"PERL_ULONG_MIN"

"PERL_USHORT_MAX"

"PERL_USHORT_MIN"

"PERL_QUAD_MAX"

"PERL_QUAD_MIN"

"PERL_UQUAD_MAX"

"PERL_UQUAD_MIN"

These give the largest and smallest number representable in the current platform in variables of the corresponding types.

For signed types, the smallest representable number is the most negative number, the one furthest away from zero.

For C99 and later compilers, these correspond to things like "INT_MAX", which are available to the C code. But these constants, furnished by Perl, allow code compiled on earlier compilers to portably have access to the same constants.

"SHORTSIZE"

This symbol contains the value of "sizeof(short)" so that the C preprocessor can make decisions based on it.

"STRLEN"

Described in perlguts.

"U8"

"U16"

"U32"

"U64"

"UV"

Described in perlguts.

"U32SIZE"

This symbol contains the "sizeof(U32)".

"U32TYPE"

This symbol defines the C type used for Perl's U32.

"U64SIZE"

This symbol contains the "sizeof(U64)".

"U64TYPE"

This symbol defines the C type used for Perl's U64.

"U16SIZE"

This symbol contains the "sizeof(U16)".

"U16TYPE"

This symbol defines the C type used for Perl's U16.

"UINT16_C"

"UINT32_C"

"UINT64_C"

Returns a token the C compiler recognizes for the constant "number" of the corresponding unsigned integer type on the machine.

If the machine does not have a 64-bit type, "UINT64_C" is undefined. Use "UINTMAX_C" to get the largest type available on the platform.

```
U16  UINT16_C(number)
U32  UINT32_C(number)
U64  UINT64_C(number)
```

"UINTMAX_C"

Returns a token the C compiler recognizes for the constant "number" of the widest unsigned integer type on the machine. For example, if the machine has "long"s, UINTMAX_C(1) would yield

```
1UL
```

See also, for example, "UINT32_C".

Use "UV" to declare variables of the maximum usable size on this platform.

```
UINTMAX_C(number)
```

"U8SIZE"

This symbol contains the "sizeof(U8)".

"U8TYPE"

This symbol defines the C type used for Perl's U8.

"UV_MAX"

The largest unsigned integer that fits in a UV on this platform.

```
UV  UV_MAX
```

"UV_MIN"

The smallest unsigned integer that fits in a UV on this platform. It should equal zero.

```
UV  UV_MIN
```

"UVSIZE"

This symbol contains the "sizeof(UV)".

"UVTYPE"

This symbol defines the C type used for Perl's UV.

"WIDEST_UTYPE"

Yields the widest unsigned integer type on the platform, currently either "U32" or "U64". This can be used in declarations such as

```
WIDEST_UTYPE my_uv;
```

or casts

```
my_uv = (WIDEST_UTYPE) val;
```

Lexer interface

This is the lower layer of the Perl parser, managing characters and tokens.

"lex_bufutf8"

NOTE: "lex_bufutf8" is experimental and may change or be removed without notice.

Indicates whether the octets in the lexer buffer ("PL_parser->linestr") should be interpreted as the UTF-8 encoding of Unicode characters. If not, they should be interpreted as Latin-1 characters. This is analogous to the "SvUTF8" flag for scalars.

In UTF-8 mode, it is not guaranteed that the lexer buffer actually contains valid UTF-8. Lexing code must be robust in the face of invalid encoding.

The actual "SvUTF8" flag of the "PL_parser->linestr" scalar is significant, but not the whole story regarding the input character encoding. Normally, when a file is being read, the scalar contains octets and its "SvUTF8" flag is off, but the octets should be interpreted as UTF-8 if the "use utf8" pragma is in effect. During a string eval, however, the scalar may have the "SvUTF8" flag on, and in this case its octets should be interpreted as UTF-8 unless the "use bytes" pragma is in effect. This logic may change in the future; use this function instead of implementing the logic yourself.

```
bool  lex_bufutf8()
```

"lex_discard_to"

NOTE: "lex_discard_to" is experimental and may change or be removed without notice.

Discards the first part of the "PL_parser->linestr" buffer, up to "ptr". The remaining content of the buffer will be moved, and all pointers into the buffer updated appropriately. "ptr" must not be later in the buffer than the position of "PL_parser->bufptr": it is not permitted to discard text that has yet to be lexed.

Normally it is not necessarily to do this directly, because it suffices to use the implicit discarding behaviour of "lex_next_chunk" and things based on it. However, if a token stretches across multiple lines, and the lexing code has kept multiple lines of text in the buffer for that purpose, then after completion of the token it would be wise to explicitly discard the now-unneeded earlier lines, to avoid future multi-line tokens growing the buffer without bound.

```
void  lex_discard_to(char* ptr)
```

"lex_grow_linestr"

NOTE: "lex_grow_linestr" is experimental and may change or be removed without notice.

Reallocates the lexer buffer ("PL_parser->linestr") to accommodate at least "len" octets (including terminating "NUL"). Returns a pointer to the reallocated buffer.

This is necessary before making any direct modification of the buffer that would increase its length. "lex_stuff_pvn" provides a more convenient way to insert text into the buffer.

Do not use "SvGROW" or "sv_grow" directly on "PL_parser->linestr"; this function updates all of the lexer's variables that point directly into the buffer.

```
char*  lex_grow_linestr(STRLEN len)
```

"lex_next_chunk"

NOTE: "lex_next_chunk" is experimental and may change or be removed without notice.

Reads in the next chunk of text to be lexed, appending it to "PL_parser->linestr". This should be called when lexing code has looked to the end of the current chunk and wants to know more. It is usual, but not necessary, for lexing to have consumed the entirety of the current chunk at this time.

If "PL_parser->bufptr" is pointing to the very end of the current chunk (i.e., the current chunk has been entirely consumed), normally the current chunk will be discarded at the same time that the new chunk is read in. If "flags" has the "LEX_KEEP_PREVIOUS" bit set, the current chunk will not be discarded. If the current chunk has not been entirely consumed, then it will not be discarded regardless of the flag.

Returns true if some new text was added to the buffer, or false if the buffer has reached the end of the input text.

```
bool  lex_next_chunk(U32 flags)
```

"lex_peek_unichar"

NOTE: "lex_peek_unichar" is experimental and may change or be removed without notice.

Looks ahead one (Unicode) character in the text currently being lexed. Returns the codepoint (unsigned integer value) of the next character, or -1 if lexing has reached the end of the input text. To consume the peeked character, use "lex_read_unichar". If the next character is in (or extends into) the next chunk of input text, the next chunk will be read in. Normally the current chunk will be discarded at the same time, but if "flags" has the "LEX_KEEP_PREVIOUS" bit set, then the current chunk will not be discarded.

If the input is being interpreted as UTF-8 and a UTF-8 encoding error is encountered, an exception is generated.

```
I32  lex_peek_unichar(U32 flags)
```

"lex_read_space"

NOTE: "lex_read_space" is experimental and may change or be removed without notice.

Reads optional spaces, in Perl style, in the text currently being lexed. The spaces may include ordinary whitespace characters and Perl-style comments. "#line" directives are processed if encountered. "PL_parser->bufptr" is moved past the spaces, so that it points at a non-space character (or the end of the input text).

If spaces extend into the next chunk of input text, the next chunk will be read in. Normally the current chunk will be discarded at the same time, but if "flags" has the "LEX_KEEP_PREVIOUS" bit set, then the current chunk will not be discarded.

```
void  lex_read_space(U32 flags)
```

"lex_read_to"

NOTE: "lex_read_to" is experimental and may change or be removed without notice.

Consume text in the lexer buffer, from "PL_parser->bufptr" up to "ptr". This advances "PL_parser->bufptr" to match "ptr", performing the correct bookkeeping whenever a newline character is passed. This is the normal way to consume lexed text.

Interpretation of the buffer's octets can be abstracted out by using the slightly higher-level functions "lex_peek_unichar" and "lex_read_unichar".

```
void  lex_read_to(char* ptr)
```

"lex_read_unichar"

NOTE: "lex_read_unichar" is experimental and may change or be removed without notice.

Reads the next (Unicode) character in the text currently being lexed. Returns the codepoint (unsigned integer value) of the character read, and moves

"PL_parser->bufptr" past the character, or returns -1 if lexing has reached the end of the input text. To non-destructively examine the next character, use "lex_peek_unichar" instead.

If the next character is in (or extends into) the next chunk of input text, the next chunk will be read in. Normally the current chunk will be discarded at the same time, but if "flags" has the "LEX_KEEP_PREVIOUS" bit set, then the current chunk will not be discarded.

If the input is being interpreted as UTF-8 and a UTF-8 encoding error is encountered, an exception is generated.

```
I32  lex_read_unichar(U32 flags)
```

"lex_start"

NOTE: "lex_start" is experimental and may change or be removed without notice.

Creates and initialises a new lexer/parser state object, supplying a context in which to lex and parse from a new source of Perl code. A pointer to the new state object is placed in "PL_parser". An entry is made on the save stack so that upon unwinding, the new state object will be destroyed and the former value of "PL_parser" will be restored. Nothing else need be done to clean up the parsing context.

The code to be parsed comes from "line" and "rsfp". "line", if non-null, provides a string (in SV form) containing code to be parsed. A copy of the string is made, so subsequent modification of "line" does not affect parsing. "rsfp", if non-null, provides an input stream from which code will be read to be parsed. If both are non-null, the code in "line" comes first and must consist of complete lines of input, and "rsfp" supplies the remainder of the source.

The "flags" parameter is reserved for future use. Currently it is only used by perl internally, so extensions should always pass zero.

```
void  lex_start(SV* line, PerlIO *rsfp, U32 flags)
```

"lex_stuff_pv"

NOTE: "lex_stuff_pv" is experimental and may change or be removed without notice.

Insert characters into the lexer buffer ("PL_parser->linestr"), immediately after the current lexing point ("PL_parser->bufptr"), reallocating the buffer if necessary.

This means that lexing code that runs later will see the characters as if they had appeared in the input. It is not recommended to do this as part of normal parsing, and most uses of this facility run the risk of the inserted characters being

interpreted in an unintended manner.

The string to be inserted is represented by octets starting at "pv" and continuing to the first nul. These octets are interpreted as either UTF-8 or Latin-1, according to whether the "LEX_STUFF_UTF8" flag is set in "flags". The characters are recoded for the lexer buffer, according to how the buffer is currently being interpreted ("lex_bufutf8"). If it is not convenient to nul-terminate a string to be inserted, the "lex_stuff_pvn" function is more appropriate.

```
void  lex_stuff_pv(const char* pv, U32 flags)
```

"lex_stuff_pvn"

NOTE: "lex_stuff_pvn" is experimental and may change or be removed without notice.

Insert characters into the lexer buffer ("PL_parser->linestr"), immediately after the current lexing point ("PL_parser->bufptr"), reallocating the buffer if necessary. This means that lexing code that runs later will see the characters as if they had appeared in the input. It is not recommended to do this as part of normal parsing, and most uses of this facility run the risk of the inserted characters being interpreted in an unintended manner.

The string to be inserted is represented by "len" octets starting at "pv". These octets are interpreted as either UTF-8 or Latin-1, according to whether the "LEX_STUFF_UTF8" flag is set in "flags". The characters are recoded for the lexer buffer, according to how the buffer is currently being interpreted ("lex_bufutf8"). If a string to be inserted is available as a Perl scalar, the "lex_stuff_sv" function is more convenient.

```
void  lex_stuff_pvn(const char* pv, STRLEN len, U32 flags)
```

"lex_stuff_pvs"

NOTE: "lex_stuff_pvs" is experimental and may change or be removed without notice.

Like "lex_stuff_pvn", but takes a literal string instead of a string/length pair.

```
void  lex_stuff_pvs("pv", U32 flags)
```

"lex_stuff_sv"

NOTE: "lex_stuff_sv" is experimental and may change or be removed without notice.

Insert characters into the lexer buffer ("PL_parser->linestr"), immediately after the current lexing point ("PL_parser->bufptr"), reallocating the buffer if necessary. This means that lexing code that runs later will see the characters as if they had appeared in the input. It is not recommended to do this as part of normal parsing,

and most uses of this facility run the risk of the inserted characters being interpreted in an unintended manner.

The string to be inserted is the string value of "sv". The characters are recoded for the lexer buffer, according to how the buffer is currently being interpreted ("lex_bufutf8"). If a string to be inserted is not already a Perl scalar, the "lex_stuff_pvn" function avoids the need to construct a scalar.

```
void  lex_stuff_sv(SV* sv, U32 flags)
```

"lex_unstuff"

NOTE: "lex_unstuff" is experimental and may change or be removed without notice.

Discards text about to be lexed, from "PL_parser->bufptr" up to "ptr". Text following "ptr" will be moved, and the buffer shortened. This hides the discarded text from any lexing code that runs later, as if the text had never appeared.

This is not the normal way to consume lexed text. For that, use "lex_read_to".

```
void  lex_unstuff(char* ptr)
```

"parse_arithexpr"

NOTE: "parse_arithexpr" is experimental and may change or be removed without notice.

Parse a Perl arithmetic expression. This may contain operators of precedence down to the bit shift operators. The expression must be followed (and thus terminated) either by a comparison or lower-precedence operator or by something that would normally terminate an expression such as semicolon. If "flags" has the "PARSE_OPTIONAL" bit set, then the expression is optional, otherwise it is mandatory. It is up to the caller to ensure that the dynamic parser state ("PL_parser" et al) is correctly set to reflect the source of the code to be parsed and the lexical context for the expression.

The op tree representing the expression is returned. If an optional expression is absent, a null pointer is returned, otherwise the pointer will be non-null.

If an error occurs in parsing or compilation, in most cases a valid op tree is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. Some compilation errors, however, will throw an exception immediately.

```
OP*  parse_arithexpr(U32 flags)
```

"parse_barestmt"

NOTE: "parse_barestmt" is experimental and may change or be removed without notice.

Parse a single unadorned Perl statement. This may be a normal imperative statement or a declaration that has compile-time effect. It does not include any label or other affixture. It is up to the caller to ensure that the dynamic parser state ("PL_parser" et al) is correctly set to reflect the source of the code to be parsed and the lexical context for the statement.

The op tree representing the statement is returned. This may be a null pointer if the statement is null, for example if it was actually a subroutine definition (which has compile-time side effects). If not null, it will be ops directly implementing the statement, suitable to pass to "newSTATEOP". It will not normally include a "nextstate" or equivalent op (except for those embedded in a scope contained entirely within the statement).

If an error occurs in parsing or compilation, in most cases a valid op tree (most likely null) is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. Some compilation errors, however, will throw an exception immediately.

The "flags" parameter is reserved for future use, and must always be zero.

OP* parse_barestmt(U32 flags)

"parse_block"

NOTE: "parse_block" is experimental and may change or be removed without notice.

Parse a single complete Perl code block. This consists of an opening brace, a sequence of statements, and a closing brace. The block constitutes a lexical scope, so "my" variables and various compile-time effects can be contained within it. It is up to the caller to ensure that the dynamic parser state ("PL_parser" et al) is correctly set to reflect the source of the code to be parsed and the lexical context for the statement.

The op tree representing the code block is returned. This is always a real op, never a null pointer. It will normally be a "lineseq" list, including "nextstate" or equivalent ops. No ops to construct any kind of runtime scope are included by virtue of it being a block.

If an error occurs in parsing or compilation, in most cases a valid op tree (most likely null) is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the

compilation errors that occurred. Some compilation errors, however, will throw an exception immediately.

The "flags" parameter is reserved for future use, and must always be zero.

OP* parse_block(U32 flags)

"parse_fullexpr"

NOTE: "parse_fullexpr" is experimental and may change or be removed without notice.

Parse a single complete Perl expression. This allows the full expression grammar, including the lowest-precedence operators such as "or". The expression must be followed (and thus terminated) by a token that an expression would normally be terminated by: end-of-file, closing bracketing punctuation, semicolon, or one of the keywords that signals a postfix expression-statement modifier. If "flags" has the "PARSE_OPTIONAL" bit set, then the expression is optional, otherwise it is mandatory. It is up to the caller to ensure that the dynamic parser state ("PL_parser" et al) is correctly set to reflect the source of the code to be parsed and the lexical context for the expression.

The op tree representing the expression is returned. If an optional expression is absent, a null pointer is returned, otherwise the pointer will be non-null.

If an error occurs in parsing or compilation, in most cases a valid op tree is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. Some compilation errors, however, will throw an exception immediately.

OP* parse_fullexpr(U32 flags)

"parse_fullstmt"

NOTE: "parse_fullstmt" is experimental and may change or be removed without notice.

Parse a single complete Perl statement. This may be a normal imperative statement or a declaration that has compile-time effect, and may include optional labels. It is up to the caller to ensure that the dynamic parser state ("PL_parser" et al) is correctly set to reflect the source of the code to be parsed and the lexical context for the statement.

The op tree representing the statement is returned. This may be a null pointer if the statement is null, for example if it was actually a subroutine definition (which has compile-time side effects). If not null, it will be the result of a "newSTATEOP" call, normally including a "nextstate" or equivalent op.

If an error occurs in parsing or compilation, in most cases a valid op tree (most likely null) is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. Some compilation errors, however, will throw an exception immediately.

The "flags" parameter is reserved for future use, and must always be zero.

OP* parse_fullstmt(U32 flags)

"parse_label"

NOTE: "parse_label" is experimental and may change or be removed without notice.

Parse a single label, possibly optional, of the type that may prefix a Perl statement. It is up to the caller to ensure that the dynamic parser state ("PL_parser" et al) is correctly set to reflect the source of the code to be parsed. If "flags" has the "PARSE_OPTIONAL" bit set, then the label is optional, otherwise it is mandatory.

The name of the label is returned in the form of a fresh scalar. If an optional label is absent, a null pointer is returned.

If an error occurs in parsing, which can only occur if the label is mandatory, a valid label is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred.

SV* parse_label(U32 flags)

"parse_listexpr"

NOTE: "parse_listexpr" is experimental and may change or be removed without notice.

Parse a Perl list expression. This may contain operators of precedence down to the comma operator. The expression must be followed (and thus terminated) either by a low-precedence logic operator such as "or" or by something that would normally terminate an expression such as semicolon. If "flags" has the "PARSE_OPTIONAL" bit set, then the expression is optional, otherwise it is mandatory. It is up to the caller to ensure that the dynamic parser state ("PL_parser" et al) is correctly set to reflect the source of the code to be parsed and the lexical context for the expression.

The op tree representing the expression is returned. If an optional expression is absent, a null pointer is returned, otherwise the pointer will be non-null.

If an error occurs in parsing or compilation, in most cases a valid op tree is

returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. Some compilation errors, however, will throw an exception immediately.

OP* parse_listexpr(U32 flags)

"parse_stmtseq"

NOTE: "parse_stmtseq" is experimental and may change or be removed without notice.

Parse a sequence of zero or more Perl statements. These may be normal imperative statements, including optional labels, or declarations that have compile-time effect, or any mixture thereof. The statement sequence ends when a closing brace or end-of-file is encountered in a place where a new statement could have validly started. It is up to the caller to ensure that the dynamic parser state ("PL_parser" et al) is correctly set to reflect the source of the code to be parsed and the lexical context for the statements.

The op tree representing the statement sequence is returned. This may be a null pointer if the statements were all null, for example if there were no statements or if there were only subroutine definitions (which have compile-time side effects). If not null, it will be a "lineseq" list, normally including "nextstate" or equivalent ops.

If an error occurs in parsing or compilation, in most cases a valid op tree is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. Some compilation errors, however, will throw an exception immediately.

The "flags" parameter is reserved for future use, and must always be zero.

OP* parse_stmtseq(U32 flags)

"parse_subsignature"

NOTE: "parse_subsignature" is experimental and may change or be removed without notice.

Parse a subroutine signature declaration. This is the contents of the parentheses following a named or anonymous subroutine declaration when the "signatures" feature is enabled. Note that this function neither expects nor consumes the opening and closing parentheses around the signature; it is the caller's job to handle these.

This function must only be called during parsing of a subroutine; after "start_subparse" has been called. It might allocate lexical variables on the pad for the current subroutine.

The op tree to unpack the arguments from the stack at runtime is returned. This op tree should appear at the beginning of the compiled function. The caller may wish to use "op_append_list" to build their function body after it, or splice it together with the body before calling "newATTRSUB".

The "flags" parameter is reserved for future use, and must always be zero.

OP* parse_subsignature(U32 flags)

"parse_termexpr"

NOTE: "parse_termexpr" is experimental and may change or be removed without notice.

Parse a Perl term expression. This may contain operators of precedence down to the assignment operators. The expression must be followed (and thus terminated) either by a comma or lower-precedence operator or by something that would normally terminate an expression such as semicolon. If "flags" has the "PARSE_OPTIONAL" bit set, then the expression is optional, otherwise it is mandatory. It is up to the caller to ensure that the dynamic parser state ("PL_parser" et al) is correctly set to reflect the source of the code to be parsed and the lexical context for the expression.

The op tree representing the expression is returned. If an optional expression is absent, a null pointer is returned, otherwise the pointer will be non-null.

If an error occurs in parsing or compilation, in most cases a valid op tree is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. Some compilation errors, however, will throw an exception immediately.

OP* parse_termexpr(U32 flags)

"PL_parser"

Pointer to a structure encapsulating the state of the parsing operation currently in progress. The pointer can be locally changed to perform a nested parse without interfering with the state of an outer parse. Individual members of "PL_parser" have their own documentation.

"PL_parser->bufend"

NOTE: "PL_parser->bufend" is experimental and may change or be removed without notice.

Direct pointer to the end of the chunk of text currently being lexed, the end of the lexer buffer. This is equal to "SvPVX(PL_parser->linestr) + SvCUR(PL_parser->linestr)". A "NUL" character (zero octet) is always located at the end of the buffer, and does not count as part of the buffer's contents.

"PL_parser->bufptr"

NOTE: "PL_parser->bufptr" is experimental and may change or be removed without notice.

Points to the current position of lexing inside the lexer buffer. Characters around this point may be freely examined, within the range delimited by "SvPVX("PL_parser->linestr")" and "PL_parser->bufend". The octets of the buffer may be intended to be interpreted as either UTF-8 or Latin-1, as indicated by "lex_bufutf8".

Lexing code (whether in the Perl core or not) moves this pointer past the characters that it consumes. It is also expected to perform some bookkeeping whenever a newline character is consumed. This movement can be more conveniently performed by the function "lex_read_to", which handles newlines appropriately.

Interpretation of the buffer's octets can be abstracted out by using the slightly higher-level functions "lex_peek_unichar" and "lex_read_unichar".

"PL_parser->linestart"

NOTE: "PL_parser->linestart" is experimental and may change or be removed without notice.

Points to the start of the current line inside the lexer buffer. This is useful for indicating at which column an error occurred, and not much else. This must be updated by any lexing code that consumes a newline; the function "lex_read_to" handles this detail.

"PL_parser->linestr"

NOTE: "PL_parser->linestr" is experimental and may change or be removed without notice.

Buffer scalar containing the chunk currently under consideration of the text currently being lexed. This is always a plain string scalar (for which "SvPOK" is true). It is not intended to be used as a scalar by normal scalar means; instead refer to the buffer directly by the pointer variables described below.

The lexer maintains various "char*" pointers to things in the "PL_parser->linestr" buffer. If "PL_parser->linestr" is ever reallocated, all of these pointers must be updated. Don't attempt to do this manually, but rather use "lex_grow_linestr" if you need to reallocate the buffer.

The content of the text chunk in the buffer is commonly exactly one complete line of input, up to and including a newline terminator, but there are situations where it is

otherwise. The octets of the buffer may be intended to be interpreted as either UTF-8 or Latin-1. The function "lex_bufutf8" tells you which. Do not use the "SvUTF8" flag on this scalar, which may disagree with it.

For direct examination of the buffer, the variable "PL_parser->bufend" points to the end of the buffer. The current lexing position is pointed to by "PL_parser->bufptr". Direct use of these pointers is usually preferable to examination of the scalar through normal scalar means.

"wrap_keyword_plugin"

NOTE: "wrap_keyword_plugin" is experimental and may change or be removed without notice.

Puts a C function into the chain of keyword plugins. This is the preferred way to manipulate the "PL_keyword_plugin" variable. "new_plugin" is a pointer to the C function that is to be added to the keyword plugin chain, and "old_plugin_p" points to the storage location where a pointer to the next function in the chain will be stored. The value of "new_plugin" is written into the "PL_keyword_plugin" variable, while the value previously stored there is written to *old_plugin_p.

"PL_keyword_plugin" is global to an entire process, and a module wishing to hook keyword parsing may find itself invoked more than once per process, typically in different threads. To handle that situation, this function is idempotent. The location *old_plugin_p must initially (once per process) contain a null pointer. A C variable of static duration (declared at file scope, typically also marked "static" to give it internal linkage) will be implicitly initialised appropriately, if it does not have an explicit initialiser. This function will only actually modify the plugin chain if it finds *old_plugin_p to be null. This function is also thread safe on the small scale. It uses appropriate locking to avoid race conditions in accessing "PL_keyword_plugin".

When this function is called, the function referenced by "new_plugin" must be ready to be called, except for *old_plugin_p being unfilled. In a threading situation, "new_plugin" may be called immediately, even before this function has returned. *old_plugin_p will always be appropriately set before "new_plugin" is called. If "new_plugin" decides not to do anything special with the identifier that it is given (which is the usual case for most calls to a keyword plugin), it must chain the plugin function referenced by *old_plugin_p.

Taken all together, XS code to install a keyword plugin should typically look something like this:

```
static Perl_keyword_plugin_t next_keyword_plugin;
static OP *my_keyword_plugin(pTHX_
    char *keyword_ptr, STRLEN keyword_len, OP **op_ptr)
{
    if (memEQs(keyword_ptr, keyword_len,
               "my_new_keyword")) {
        ...
    } else {
        return next_keyword_plugin(aTHX_
            keyword_ptr, keyword_len, op_ptr);
    }
}
BOOT:
    wrap_keyword_plugin(my_keyword_plugin,
                        &next_keyword_plugin);
```

Direct access to "PL_keyword_plugin" should be avoided.

```
 void  wrap_keyword_plugin(Perl_keyword_plugin_t new_plugin,
                           Perl_keyword_plugin_t *old_plugin_p)
```

Locales

"DECLARATION_FOR_LC_NUMERIC_MANIPULATION"

This macro should be used as a statement. It declares a private variable (whose name begins with an underscore) that is needed by the other macros in this section. Failing to include this correctly should lead to a syntax error. For compatibility with C89 C compilers it should be placed in a block before any executable statements.

```
 void  DECLARATION_FOR_LC_NUMERIC_MANIPULATION
```

"foldEQ_locale"

Returns true if the leading "len" bytes of the strings "s1" and "s2" are the same case-insensitively in the current locale; false otherwise.

```
 I32  foldEQ_locale(const char* a, const char* b, I32 len)
```

"HAS_DUPLOCALE"

This symbol, if defined, indicates that the "duplocale" routine is available to

duplicate a locale object.

"HAS_FREELOCALE"

This symbol, if defined, indicates that the "freelocale" routine is available to deallocates the resources associated with a locale object.

"HAS_LC_MONETARY_2008"

This symbol, if defined, indicates that the localeconv routine is available and has the additional members added in "POSIX" 1003.1-2008.

"HAS_LOCALECONV"

This symbol, if defined, indicates that the "localeconv" routine is available for numeric and monetary formatting conventions.

"HAS_LOCALECONV_L"

This symbol, if defined, indicates that the "localeconv_l" routine is available to query certain information about a locale.

"HAS_NEWLOCALE"

This symbol, if defined, indicates that the "newlocale" routine is available to return a new locale object or modify an existing locale object.

"HAS_NL_LANGINFO"

This symbol, if defined, indicates that the "nl_langinfo" routine is available to return local data. You will also need langinfo.h and therefore "I_LANGINFO".

"HAS_QUERYLOCALE"

This symbol, if defined, indicates that the "querylocale" routine is available to return the name of the locale for a category mask.

"HAS_SETLOCALE"

This symbol, if defined, indicates that the "setlocale" routine is available to handle locale-specific ctype implementations.

"HAS_SETLOCALE_R"

This symbol, if defined, indicates that the "setlocale_r" routine is available to setlocale re-entrantly.

"HAS_THREAD_SAFE_NL_LANGINFO_L"

This symbol, when defined, indicates presence of the "nl_langinfo_l()" function, and that it is thread-safe.

"HAS_USELOCALE"

This symbol, if defined, indicates that the "uselocale" routine is available to set

the current locale for the calling thread.

"I_LANGINFO"

This symbol, if defined, indicates that langinfo.h exists and should be included.

```
#ifdef I_LANGINFO
    #include <langinfo.h>
#endif
```

"I_LOCALE"

This symbol, if defined, indicates to the C program that it should include locale.h.

```
#ifdef I_LOCALE
    #include <locale.h>
#endif
```

"IN_LOCALE"

Evaluates to TRUE if the plain locale pragma without a parameter ("use?locale") is in effect.

```
bool  IN_LOCALE
```

"IN_LOCALE_COMPILETIME"

Evaluates to TRUE if, when compiling a perl program (including an "eval") if the plain locale pragma without a parameter ("use?locale") is in effect.

```
bool  IN_LOCALE_COMPILETIME
```

"IN_LOCALE_RUNTIME"

Evaluates to TRUE if, when executing a perl program (including an "eval") if the plain locale pragma without a parameter ("use?locale") is in effect.

```
bool  IN_LOCALE_RUNTIME
```

"I_XLOCALE"

This symbol, if defined, indicates to the C program that it should include xlocale.h to get "uselocale()" and its friends.

```
#ifdef I_XLOCALE
    #include <xlocale.h>
#endif
```

"Perl_langinfo"

This is an (almost) drop-in replacement for the system nl_langinfo(3), taking the same "item" parameter values, and returning the same information. But it is more thread-safe than regular "nl_langinfo()", and hides the quirks of Perl's locale handling from

your code, and can be used on systems that lack a native "nl_langinfo".

Expanding on these:

? The reason it isn't quite a drop-in replacement is actually an advantage. The only difference is that it returns "const?char?*", whereas plain "nl_langinfo()" returns "char?*", but you are (only by documentation) forbidden to write into the buffer. By declaring this "const", the compiler enforces this restriction, so if it is violated, you know at compilation time, rather than getting segfaults at runtime.

? It delivers the correct results for the "RADIXCHAR" and "THOUSEP" items, without you having to write extra code. The reason for the extra code would be because these are from the "LC_NUMERIC" locale category, which is normally kept set by Perl so that the radix is a dot, and the separator is the empty string, no matter what the underlying locale is supposed to be, and so to get the expected results, you have to temporarily toggle into the underlying locale, and later toggle back. (You could use plain "nl_langinfo" and "STORE_LC_NUMERIC_FORCE_TO_UNDERLYING" for this but then you wouldn't get the other advantages of "Perl_langinfo()"; not keeping "LC_NUMERIC" in the C (or equivalent) locale would break a lot of CPAN, which is expecting the radix (decimal point) character to be a dot.)

? The system function it replaces can have its static return buffer trashed, not only by a subsequent call to that function, but by a "freelocale", "setlocale", or other locale change. The returned buffer of this function is not changed until the next call to it, so the buffer is never in a trashed state.

? Its return buffer is per-thread, so it also is never overwritten by a call to this function from another thread; unlike the function it replaces.

? But most importantly, it works on systems that don't have "nl_langinfo", such as Windows, hence makes your code more portable. Of the fifty-some possible items specified by the POSIX 2008 standard, <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/langinfo.h.html>, only one is completely unimplemented, though on non-Windows platforms, another significant one is also not implemented). It uses various techniques to recover the other items, including calling localeconv(3), and strftime(3), both of which are specified in C89, so should be always be available. Later "strftime()" versions have additional capabilities; "" is returned for those not available on

your system.

It is important to note that when called with an item that is recovered by using "localeconv", the buffer from any previous explicit call to "localeconv" will be overwritten. This means you must save that buffer's contents if you need to access them after a call to this function. (But note that you might not want to be using "localeconv()" directly anyway, because of issues like the ones listed in the second item of this list (above) for "RADIXCHAR" and "THOUSEP". You can use the methods given in perlcall to call "localeconv" in POSIX and avoid all the issues, but then you have a hash to unpack).

The details for those items which may deviate from what this emulation returns and what a native "nl_langinfo()" would return are specified in I18N::Langinfo.

When using "Perl_langinfo" on systems that don't have a native "nl_langinfo()", you must

```
#include "perl_langinfo.h"
```

before the "perl.h" "#include". You can replace your "langinfo.h" "#include" with this one. (Doing it this way keeps out the symbols that plain "langinfo.h" would try to import into the namespace for code that doesn't need it.)

The original impetus for "Perl_langinfo()" was so that code that needs to find out the current currency symbol, floating point radix character, or digit grouping separator can use, on all systems, the simpler and more thread-friendly "nl_langinfo" API instead of localeconv(3) which is a pain to make thread-friendly. For other fields returned by "localeconv", it is better to use the methods given in perlcall to call "POSIX::localeconv()", which is thread-friendly.

```
const char*  Perl_langinfo(const nl_item item)
```

"Perl_setlocale"

This is an (almost) drop-in replacement for the system setlocale(3), taking the same parameters, and returning the same information, except that it returns the correct underlying "LC_NUMERIC" locale. Regular "setlocale" will instead return "C" if the underlying locale has a non-dot decimal point character, or a non-empty thousands separator for displaying floating point numbers. This is because perl keeps that locale category such that it has a dot and empty separator, changing the locale briefly during the operations where the underlying one is required. "Perl_setlocale" knows about this, and compensates; regular "setlocale" doesn't.

Another reason it isn't completely a drop-in replacement is that it is declared to return "const?char?*", whereas the system setlocale omits the "const" (presumably because its API was specified long ago, and can't be updated; it is illegal to change the information "setlocale" returns; doing so leads to segfaults.)

Finally, "Perl_setlocale" works under all circumstances, whereas plain "setlocale" can be completely ineffective on some platforms under some configurations.

"Perl_setlocale" should not be used to change the locale except on systems where the predefined variable "${^SAFE_LOCALES}" is 1. On some such systems, the system "setlocale()" is ineffective, returning the wrong information, and failing to actually change the locale. "Perl_setlocale", however works properly in all circumstances.

The return points to a per-thread static buffer, which is overwritten the next time "Perl_setlocale" is called from the same thread.

```
const char*  Perl_setlocale(const int category,
                            const char* locale)
```

"RESTORE_LC_NUMERIC"

This is used in conjunction with one of the macros "STORE_LC_NUMERIC_SET_TO_NEEDED" and "STORE_LC_NUMERIC_FORCE_TO_UNDERLYING" to properly restore the "LC_NUMERIC" state.

A call to "DECLARATION_FOR_LC_NUMERIC_MANIPULATION" must have been made to declare at compile time a private variable used by this macro and the two "STORE" ones. This macro should be called as a single statement, not an expression, but with an empty argument list, like this:

```
{
    DECLARATION_FOR_LC_NUMERIC_MANIPULATION;
     ...
    RESTORE_LC_NUMERIC();
     ...
}
```

```
void  RESTORE_LC_NUMERIC()
```

"SETLOCALE_ACCEPTS_ANY_LOCALE_NAME"

This symbol, if defined, indicates that the setlocale routine is available and it accepts any input locale name as valid.

"STORE_LC_NUMERIC_FORCE_TO_UNDERLYING"

This is used by XS code that is "LC_NUMERIC" locale-aware to force the locale for

category "LC_NUMERIC" to be what perl thinks is the current underlying locale. (The perl interpreter could be wrong about what the underlying locale actually is if some C or XS code has called the C library function setlocale(3) behind its back; calling "sync_locale" before calling this macro will update perl's records.)

A call to "DECLARATION_FOR_LC_NUMERIC_MANIPULATION" must have been made to declare at compile time a private variable used by this macro. This macro should be called as a single statement, not an expression, but with an empty argument list, like this:

```
{
    DECLARATION_FOR_LC_NUMERIC_MANIPULATION;
     ...
    STORE_LC_NUMERIC_FORCE_TO_UNDERLYING();
     ...
    RESTORE_LC_NUMERIC();
     ...
}
```

The private variable is used to save the current locale state, so that the requisite matching call to "RESTORE_LC_NUMERIC" can restore it.

On threaded perls not operating with thread-safe functionality, this macro uses a mutex to force a critical section. Therefore the matching RESTORE should be close by, and guaranteed to be called.

```
 void  STORE_LC_NUMERIC_FORCE_TO_UNDERLYING()
```

"STORE_LC_NUMERIC_SET_TO_NEEDED"

This is used to help wrap XS or C code that is "LC_NUMERIC" locale-aware. This locale category is generally kept set to a locale where the decimal radix character is a dot, and the separator between groups of digits is empty. This is because most XS code that reads floating point numbers is expecting them to have this syntax.

This macro makes sure the current "LC_NUMERIC" state is set properly, to be aware of locale if the call to the XS or C code from the Perl program is from within the scope of a "use?locale"; or to ignore locale if the call is instead from outside such scope.

This macro is the start of wrapping the C or XS code; the wrap ending is done by calling the "RESTORE_LC_NUMERIC" macro after the operation. Otherwise the state can be changed that will adversely affect other XS code.

A call to "DECLARATION_FOR_LC_NUMERIC_MANIPULATION" must have been made to declare at

compile time a private variable used by this macro. This macro should be called as a single statement, not an expression, but with an empty argument list, like this:

```
{
    DECLARATION_FOR_LC_NUMERIC_MANIPULATION;
    ...
    STORE_LC_NUMERIC_SET_TO_NEEDED();
    ...
    RESTORE_LC_NUMERIC();
    ...
}
```

On threaded perls not operating with thread-safe functionality, this macro uses a mutex to force a critical section. Therefore the matching RESTORE should be close by, and guaranteed to be called; see "WITH_LC_NUMERIC_SET_TO_NEEDED" for a more contained way to ensure that.

void STORE_LC_NUMERIC_SET_TO_NEEDED()

"STORE_LC_NUMERIC_SET_TO_NEEDED_IN"

Same as "STORE_LC_NUMERIC_SET_TO_NEEDED" with in_lc_numeric provided as the precalculated value of "IN_LC(LC_NUMERIC)". It is the caller's responsibility to ensure that the status of "PL_compiling" and "PL_hints" cannot have changed since the precalculation.

void STORE_LC_NUMERIC_SET_TO_NEEDED_IN(bool in_lc_numeric)

"switch_to_global_locale"

On systems without locale support, or on typical single-threaded builds, or on platforms that do not support per-thread locale operations, this function does nothing. On such systems that do have locale support, only a locale global to the whole program is available.

On multi-threaded builds on systems that do have per-thread locale operations, this function converts the thread it is running in to use the global locale. This is for code that has not yet or cannot be updated to handle multi-threaded locale operation. As long as only a single thread is so-converted, everything works fine, as all the other threads continue to ignore the global one, so only this thread looks at it.

However, on Windows systems this isn't quite true prior to Visual Studio 15, at which point Microsoft fixed a bug. A race can occur if you use the following operations on

earlier Windows platforms:

POSIX::localeconv

I18N::Langinfo, items "CRNCYSTR" and "THOUSEP"

"Perl_langinfo" in perlapi, items "CRNCYSTR" and "THOUSEP"

The first item is not fixable (except by upgrading to a later Visual Studio release), but it would be possible to work around the latter two items by using the Windows API functions "GetNumberFormat" and "GetCurrencyFormat"; patches welcome.

Without this function call, threads that use the setlocale(3) system function will not work properly, as all the locale-sensitive functions will look at the per-thread locale, and "setlocale" will have no effect on this thread.

Perl code should convert to either call "Perl_setlocale" (which is a drop-in for the system "setlocale") or use the methods given in perlcall to call "POSIX::setlocale". Either one will transparently properly handle all cases of single- vs multi-thread, POSIX 2008-supported or not.

Non-Perl libraries, such as "gtk", that call the system "setlocale" can continue to work if this function is called before transferring control to the library.

Upon return from the code that needs to use the global locale, "sync_locale()" should be called to restore the safe multi-thread operation.

```
 void  switch_to_global_locale()
```

"sync_locale"

"Perl_setlocale" can be used at any time to query or change the locale (though changing the locale is antisocial and dangerous on multi-threaded systems that don't have multi-thread safe locale operations. (See "Multi-threaded operation" in perllocale). Using the system setlocale(3) should be avoided. Nevertheless, certain non-Perl libraries called from XS, such as "Gtk" do so, and this can't be changed. When the locale is changed by XS code that didn't use "Perl_setlocale", Perl needs to be told that the locale has changed. Use this function to do so, before returning to Perl.

The return value is a boolean: TRUE if the global locale at the time of call was in effect; and FALSE if a per-thread locale was in effect. This can be used by the caller that needs to restore things as-they-were to decide whether or not to call "Perl_switch_to_global_locale".

```
 bool  sync_locale()
```

"WITH_LC_NUMERIC_SET_TO_NEEDED"

This macro invokes the supplied statement or block within the context of a "STORE_LC_NUMERIC_SET_TO_NEEDED" .. "RESTORE_LC_NUMERIC" pair if required, so eg:

```
 WITH_LC_NUMERIC_SET_TO_NEEDED(
   SNPRINTF_G(fv, ebuf, sizeof(ebuf), precis)
 );
```

is equivalent to:

```
 {
#ifdef USE_LOCALE_NUMERIC
   DECLARATION_FOR_LC_NUMERIC_MANIPULATION;
   STORE_LC_NUMERIC_SET_TO_NEEDED();
#endif
   SNPRINTF_G(fv, ebuf, sizeof(ebuf), precis);
#ifdef USE_LOCALE_NUMERIC
   RESTORE_LC_NUMERIC();
#endif
 }
```

```
 void  WITH_LC_NUMERIC_SET_TO_NEEDED(block)
```

"WITH_LC_NUMERIC_SET_TO_NEEDED_IN"

Same as "WITH_LC_NUMERIC_SET_TO_NEEDED" with in_lc_numeric provided as the precalculated value of "IN_LC(LC_NUMERIC)". It is the caller's responsibility to ensure that the status of "PL_compiling" and "PL_hints" cannot have changed since the precalculation.

```
 void  WITH_LC_NUMERIC_SET_TO_NEEDED_IN(bool in_lc_numeric, block)
```

Magic

"Magic" is special data attached to SV structures in order to give them "magical" properties. When any Perl code tries to read from, or assign to, an SV marked as magical, it calls the 'get' or 'set' function associated with that SV's magic. A get is called prior to reading an SV, in order to give it a chance to update its internal value (get on $. writes the line number of the last read filehandle into the SV's IV slot), while set is called after an SV has been written to, in order to allow it to make use of its changed value (set on $/ copies the SV's new value to the PL_rs global variable).

Magic is implemented as a linked list of MAGIC structures attached to the SV. Each MAGIC

struct holds the type of the magic, a pointer to an array of functions that implement the get(), set(), length() etc functions, plus space for some flags and pointers. For example, a tied variable has a MAGIC structure that contains a pointer to the object associated with the tie.

"mg_clear"

Clear something magical that the SV represents. See "sv_magic".

```
int  mg_clear(SV* sv)
```

"mg_copy"

Copies the magic from one SV to another. See "sv_magic".

```
int  mg_copy(SV *sv, SV *nsv, const char *key, I32 klen)
```

"mg_find"

Finds the magic pointer for "type" matching the SV. See "sv_magic".

```
MAGIC*  mg_find(const SV* sv, int type)
```

"mg_findext"

Finds the magic pointer of "type" with the given "vtbl" for the "SV". See "sv_magicext".

```
MAGIC*  mg_findext(const SV* sv, int type, const MGVTBL *vtbl)
```

"mg_free"

Free any magic storage used by the SV. See "sv_magic".

```
int  mg_free(SV* sv)
```

"mg_freeext"

Remove any magic of type "how" using virtual table "vtbl" from the SV "sv". See "sv_magic".

"mg_freeext(sv, how, NULL)" is equivalent to "mg_free_type(sv, how)".

```
void  mg_freeext(SV* sv, int how, const MGVTBL *vtbl)
```

"mg_free_type"

Remove any magic of type "how" from the SV "sv". See "sv_magic".

```
void  mg_free_type(SV* sv, int how)
```

"mg_get"

Do magic before a value is retrieved from the SV. The type of SV must be >= "SVt_PVMG". See "sv_magic".

```
int  mg_get(SV* sv)
```

"mg_length"

"DEPRECATED!" It is planned to remove "mg_length" from a future release of Perl. Do not use it for new code; remove it from existing code.

Reports on the SV's length in bytes, calling length magic if available, but does not set the UTF8 flag on "sv". It will fall back to 'get' magic if there is no 'length' magic, but with no indication as to whether it called 'get' magic. It assumes "sv" is a "PVMG" or higher. Use "sv_len()" instead.

```
U32  mg_length(SV* sv)
```

"mg_magical"

Turns on the magical status of an SV. See "sv_magic".

```
void  mg_magical(SV* sv)
```

"mg_set"

Do magic after a value is assigned to the SV. See "sv_magic".

```
int  mg_set(SV* sv)
```

"SvTIED_obj"

Described in perlinterp.

```
SvTIED_obj(SV *sv, MAGIC *mg)
```

Memory Management

"HASATTRIBUTE_MALLOC"

Can we handle "GCC" attribute for malloc-style functions.

"HAS_MALLOC_GOOD_SIZE"

This symbol, if defined, indicates that the "malloc_good_size" routine is available for use.

"HAS_MALLOC_SIZE"

This symbol, if defined, indicates that the "malloc_size" routine is available for use.

"I_MALLOCMALLOC"

This symbol, if defined, indicates to the C program that it should include malloc/malloc.h.

```
#ifdef I_MALLOCMALLOC
    #include <mallocmalloc.h>
#endif
```

"MYMALLOC"

This symbol, if defined, indicates that we're using our own malloc.

"Newx"

The XSUB-writer's interface to the C "malloc" function.

Memory obtained by this should ONLY be freed with "Safefree".

In 5.9.3, Newx() and friends replace the older New() API, and drops the first parameter, x, a debug aid which allowed callers to identify themselves. This aid has been superseded by a new build option, PERL_MEM_LOG (see "PERL_MEM_LOG" in perlhacktips). The older API is still there for use in XS modules supporting older perls.

```
void  Newx(void* ptr, int nitems, type)
```

"Newxc"

The XSUB-writer's interface to the C "malloc" function, with cast. See also "Newx".

Memory obtained by this should ONLY be freed with "Safefree".

```
void  Newxc(void* ptr, int nitems, type, cast)
```

"Newxz"

The XSUB-writer's interface to the C "malloc" function. The allocated memory is zeroed with "memzero". See also "Newx".

Memory obtained by this should ONLY be freed with "Safefree".

```
void  Newxz(void* ptr, int nitems, type)
```

"PERL_MALLOC_WRAP"

This symbol, if defined, indicates that we'd like malloc wrap checks.

"Renew"

The XSUB-writer's interface to the C "realloc" function.

Memory obtained by this should ONLY be freed with "Safefree".

```
void  Renew(void* ptr, int nitems, type)
```

"Renewc"

The XSUB-writer's interface to the C "realloc" function, with cast.

Memory obtained by this should ONLY be freed with "Safefree".

```
void  Renewc(void* ptr, int nitems, type, cast)
```

"Safefree"

The XSUB-writer's interface to the C "free" function.

This should ONLY be used on memory obtained using "Newx" and friends.

```
void  Safefree(void* ptr)
```

"safesyscalloc"

Safe version of system's calloc()

Malloc_t safesyscalloc(MEM_SIZE elements, MEM_SIZE size)

"safesysfree"

Safe version of system's free()

Free_t safesysfree(Malloc_t where)

"safesysmalloc"

Paranoid version of system's malloc()

Malloc_t safesysmalloc(MEM_SIZE nbytes)

"safesysrealloc"

Paranoid version of system's realloc()

Malloc_t safesysrealloc(Malloc_t where, MEM_SIZE nbytes)

MRO

These functions are related to the method resolution order of perl classes Also see perlmroapi.

"HvMROMETA"

Described in perlmroapi.

struct mro_meta * HvMROMETA(HV *hv)

"mro_get_linear_isa"

Returns the mro linearisation for the given stash. By default, this will be whatever "mro_get_linear_isa_dfs" returns unless some other MRO is in effect for the stash.

The return value is a read-only AV*.

You are responsible for "SvREFCNT_inc()" on the return value if you plan to store it anywhere semi-permanently (otherwise it might be deleted out from under you the next time the cache is invalidated).

AV* mro_get_linear_isa(HV* stash)

"MRO_GET_PRIVATE_DATA"

Described in perlmroapi.

SV* MRO_GET_PRIVATE_DATA(struct mro_meta *const smeta,
                          const struct mro_alg *const which)

"mro_method_changed_in"

Invalidates method caching on any child classes of the given stash, so that they might notice the changes in this one.

Ideally, all instances of "PL_sub_generation++" in perl source outside of mro.c should

be replaced by calls to this.

Perl automatically handles most of the common ways a method might be redefined.

However, there are a few ways you could change a method in a stash without the cache code noticing, in which case you need to call this method afterwards:

1) Directly manipulating the stash HV entries from XS code.

2) Assigning a reference to a readonly scalar constant into a stash entry in order to create a constant subroutine (like constant.pm does).

This same method is available from pure perl via, "mro::method_changed_in(classname)".

```
 void  mro_method_changed_in(HV* stash)
```

"mro_register"

Registers a custom mro plugin. See perlmroapi for details on this and other mro functions.

NOTE: "mro_register" must be explicitly called as "Perl_mro_register" with an "aTHX_" parameter.

```
 void  Perl_mro_register(pTHX_ const struct mro_alg *mro)
```

"mro_set_private_data"

Described in perlmroapi.

NOTE: "mro_set_private_data" must be explicitly called as "Perl_mro_set_private_data" with an "aTHX_" parameter.

```
 SV*  Perl_mro_set_private_data(pTHX_
                                struct mro_meta *const smeta,
                                const struct mro_alg *const which,
                                SV *const data)
```

Multicall Functions

"dMULTICALL"

Declare local variables for a multicall. See "LIGHTWEIGHT CALLBACKS" in perlcall.

```
  dMULTICALL;
```

"MULTICALL"

Make a lightweight callback. See "LIGHTWEIGHT CALLBACKS" in perlcall.

```
  MULTICALL;
```

"POP_MULTICALL"

Closing bracket for a lightweight callback. See "LIGHTWEIGHT CALLBACKS" in perlcall.

```
  POP_MULTICALL;
```

"PUSH_MULTICALL"

Opening bracket for a lightweight callback. See "LIGHTWEIGHT CALLBACKS" in perlcall.

```
PUSH_MULTICALL(CV* the_cv);
```

Numeric Functions

"Drand01"

This macro is to be used to generate uniformly distributed random numbers over the range [0., 1.[. You may have to supply an 'extern double "drand48()";' in your program since SunOS 4.1.3 doesn't provide you with anything relevant in its headers. See "HAS_DRAND48_PROTO".

```
double  Drand01()
```

"Gconvert"

This preprocessor macro is defined to convert a floating point number to a string without a trailing decimal point. This emulates the behavior of "sprintf("%g")", but is sometimes much more efficient. If "gconvert()" is not available, but "gcvt()" drops the trailing decimal point, then "gcvt()" is used. If all else fails, a macro using "sprintf("%g")" is used. Arguments for the Gconvert macro are: value, number of digits, whether trailing zeros should be retained, and the output buffer. The usual values are:

```
d_Gconvert='gconvert((x),(n),(t),(b))'
d_Gconvert='gcvt((x),(n),(b))'
d_Gconvert='sprintf((b),"%.*g",(n),(x))'
```

The last two assume trailing zeros should not be kept.

```
char *  Gconvert(double x, Size_t n, bool t, char * b)
```

"grok_bin"

converts a string representing a binary number to numeric form.

On entry "start" and *len_p give the string to scan, *flags gives conversion flags, and "result" should be "NULL" or a pointer to an NV. The scan stops at the end of the string, or at just before the first invalid character. Unless "PERL_SCAN_SILENT_ILLDIGIT" is set in *flags, encountering an invalid character (except NUL) will also trigger a warning. On return *len_p is set to the length of the scanned string, and *flags gives output flags.

If the value is <= "UV_MAX" it is returned as a UV, the output flags are clear, and nothing is written to *result. If the value is > "UV_MAX", "grok_bin" returns

"UV_MAX", sets "PERL_SCAN_GREATER_THAN_UV_MAX" in the output flags, and writes an approximation of the correct value into *result (which is an NV; or the approximation is discarded if "result" is NULL).

The binary number may optionally be prefixed with "0b" or "b" unless "PERL_SCAN_DISALLOW_PREFIX" is set in *flags on entry.

If "PERL_SCAN_ALLOW_UNDERSCORES" is set in *flags then any or all pairs of digits may be separated from each other by a single underscore; also a single leading underscore is accepted.

```
 UV  grok_bin(const char* start, STRLEN* len_p, I32* flags,
              NV *result)
```

"grok_hex"

converts a string representing a hex number to numeric form.

On entry "start" and *len_p give the string to scan, *flags gives conversion flags, and "result" should be "NULL" or a pointer to an NV. The scan stops at the end of the string, or at just before the first invalid character. Unless "PERL_SCAN_SILENT_ILLDIGIT" is set in *flags, encountering an invalid character (except NUL) will also trigger a warning. On return *len_p is set to the length of the scanned string, and *flags gives output flags.

If the value is <= "UV_MAX" it is returned as a UV, the output flags are clear, and nothing is written to *result. If the value is > "UV_MAX", "grok_hex" returns "UV_MAX", sets "PERL_SCAN_GREATER_THAN_UV_MAX" in the output flags, and writes an approximation of the correct value into *result (which is an NV; or the approximation is discarded if "result" is NULL).

The hex number may optionally be prefixed with "0x" or "x" unless "PERL_SCAN_DISALLOW_PREFIX" is set in *flags on entry.

If "PERL_SCAN_ALLOW_UNDERSCORES" is set in *flags then any or all pairs of digits may be separated from each other by a single underscore; also a single leading underscore is accepted.

```
 UV  grok_hex(const char* start, STRLEN* len_p, I32* flags,
              NV *result)
```

"grok_infnan"

Helper for "grok_number()", accepts various ways of spelling "infinity" or "not a number", and returns one of the following flag combinations:

IS_NUMBER_INFINITY

IS_NUMBER_NAN

IS_NUMBER_INFINITY | IS_NUMBER_NEG

IS_NUMBER_NAN | IS_NUMBER_NEG

0

possibly |-ed with "IS_NUMBER_TRAILING".

If an infinity or a not-a-number is recognized, *sp will point to one byte past the end of the recognized string. If the recognition fails, zero is returned, and *sp will not move.

```
int  grok_infnan(const char** sp, const char *send)
```

"grok_number"

Identical to "grok_number_flags()" with "flags" set to zero.

```
int  grok_number(const char *pv, STRLEN len, UV *valuep)
```

"grok_number_flags"

Recognise (or not) a number. The type of the number is returned (0 if unrecognised), otherwise it is a bit-ORed combination of "IS_NUMBER_IN_UV", "IS_NUMBER_GREATER_THAN_UV_MAX", "IS_NUMBER_NOT_INT", "IS_NUMBER_NEG", "IS_NUMBER_INFINITY", "IS_NUMBER_NAN" (defined in perl.h).

If the value of the number can fit in a UV, it is returned in *valuep. "IS_NUMBER_IN_UV" will be set to indicate that *valuep is valid, "IS_NUMBER_IN_UV" will never be set unless *valuep is valid, but *valuep may have been assigned to during processing even though "IS_NUMBER_IN_UV" is not set on return. If "valuep" is "NULL", "IS_NUMBER_IN_UV" will be set for the same cases as when "valuep" is non-"NULL", but no actual assignment (or SEGV) will occur.

"IS_NUMBER_NOT_INT" will be set with "IS_NUMBER_IN_UV" if trailing decimals were seen (in which case *valuep gives the true value truncated to an integer), and "IS_NUMBER_NEG" if the number is negative (in which case *valuep holds the absolute value). "IS_NUMBER_IN_UV" is not set if "e" notation was used or the number is larger than a UV.

"flags" allows only "PERL_SCAN_TRAILING", which allows for trailing non-numeric text on an otherwise successful grok, setting "IS_NUMBER_TRAILING" on the result.

```
int  grok_number_flags(const char *pv, STRLEN len, UV *valuep,
                       U32 flags)
```

"GROK_NUMERIC_RADIX"

A synonym for "grok_numeric_radix"

```
bool  GROK_NUMERIC_RADIX(NN const char **sp, NN const char *send)
```

"grok_numeric_radix"

Scan and skip for a numeric decimal separator (radix).

```
bool  grok_numeric_radix(const char **sp, const char *send)
```

"grok_oct"

converts a string representing an octal number to numeric form.

On entry "start" and *len_p give the string to scan, *flags gives conversion flags, and "result" should be "NULL" or a pointer to an NV. The scan stops at the end of the string, or at just before the first invalid character. Unless "PERL_SCAN_SILENT_ILLDIGIT" is set in *flags, encountering an invalid character (except NUL) will also trigger a warning. On return *len_p is set to the length of the scanned string, and *flags gives output flags.

If the value is <= "UV_MAX" it is returned as a UV, the output flags are clear, and nothing is written to *result. If the value is > "UV_MAX", "grok_oct" returns "UV_MAX", sets "PERL_SCAN_GREATER_THAN_UV_MAX" in the output flags, and writes an approximation of the correct value into *result (which is an NV; or the approximation is discarded if "result" is NULL).

If "PERL_SCAN_ALLOW_UNDERSCORES" is set in *flags then any or all pairs of digits may be separated from each other by a single underscore; also a single leading underscore is accepted.

The "PERL_SCAN_DISALLOW_PREFIX" flag is always treated as being set for this function.

```
UV  grok_oct(const char* start, STRLEN* len_p, I32* flags,
             NV *result)
```

"isinfnan"

"Perl_isinfnan()" is a utility function that returns true if the NV argument is either an infinity or a "NaN", false otherwise. To test in more detail, use "Perl_isinf()" and "Perl_isnan()".

This is also the logical inverse of Perl_isfinite().

```
bool  isinfnan(NV nv)
```

"my_atof"

"atof"(3), but properly works with Perl locale handling, accepting a dot radix

character always, but also the current locale's radix character if and only if called from within the lexical scope of a Perl "use locale" statement.

N.B. "s" must be NUL terminated.

```
 NV  my_atof(const char *s)
```

"my_strtod"

This function is equivalent to the libc strtod() function, and is available even on platforms that lack plain strtod(). Its return value is the best available precision depending on platform capabilities and Configure options.

It properly handles the locale radix character, meaning it expects a dot except when called from within the scope of "use?locale", in which case the radix character should be that specified by the current locale.

The synonym Strtod() may be used instead.

```
 NV  my_strtod(const char * const s, char ** e)
```

"PERL_ABS"

Typeless "abs" or "fabs", etc. (The usage below indicates it is for integers, but it works for any type.) Use instead of these, since the C library ones force their argument to be what it is expecting, potentially leading to disaster. But also beware that this evaluates its argument twice, so no "x++".

```
 int  PERL_ABS(int x)
```

"Perl_acos"

"Perl_asin"

"Perl_atan"

"Perl_atan2"

"Perl_ceil"

"Perl_cos"

"Perl_cosh"

"Perl_exp"

"Perl_floor"

"Perl_fmod"

"Perl_frexp"

"Perl_isfinite"

"Perl_isinf"

"Perl_isnan"

"Perl_ldexp"

"Perl_log"

"Perl_log10"

"Perl_modf"

"Perl_pow"

"Perl_sin"

"Perl_sinh"

"Perl_sqrt"

"Perl_tan"

"Perl_tanh"

These perform the corresponding mathematical operation on the operand(s), using the libc function designed for the task that has just enough precision for an NV on this platform. If no such function with sufficient precision exists, the highest precision one available is used.

```
NV  Perl_acos    (NV x)
NV  Perl_asin    (NV x)
NV  Perl_atan    (NV x)
NV  Perl_atan2   (NV x, NV y)
NV  Perl_ceil    (NV x)
NV  Perl_cos     (NV x)
NV  Perl_cosh    (NV x)
NV  Perl_exp     (NV x)
NV  Perl_floor   (NV x)
NV  Perl_fmod    (NV x, NV y)
NV  Perl_frexp   (NV x, int *exp)
IV  Perl_isfinite(NV x)
IV  Perl_isinf   (NV x)
IV  Perl_isnan   (NV x)
NV  Perl_ldexp   (NV x, int exp)
NV  Perl_log     (NV x)
NV  Perl_log10   (NV x)
NV  Perl_modf    (NV x, NV *iptr)
NV  Perl_pow     (NV x, NV y)
```

NV Perl_sin   (NV x)

NV Perl_sinh   (NV x)

NV Perl_sqrt   (NV x)

NV Perl_tan   (NV x)

NV Perl_tanh   (NV x)

"Perl_signbit"

NOTE: "Perl_signbit" is experimental and may change or be removed without notice.

Return a non-zero integer if the sign bit on an NV is set, and 0 if it is not.

If Configure detects this system has a "signbit()" that will work with our NVs, then we just use it via the "#define" in perl.h. Otherwise, fall back on this implementation. The main use of this function is catching "-0.0".

"Configure" notes: This function is called 'Perl_signbit' instead of a plain 'signbit' because it is easy to imagine a system having a "signbit()" function or macro that doesn't happen to work with our particular choice of NVs. We shouldn't just re-"#define" "signbit" as "Perl_signbit" and expect the standard system headers to be happy. Also, this is a no-context function (no "pTHX_") because "Perl_signbit()" is usually re-"#defined" in perl.h as a simple macro call to the system's "signbit()". Users should just always call "Perl_signbit()".

 int Perl_signbit(NV f)

"PL_hexdigit"

This array, indexed by an integer, converts that value into the character that represents it. For example, if the input is 8, the return will be a string whose first character is '8'. What is actually returned is a pointer into a string. All you are interested in is the first character of that string. To get uppercase letters (for the values 10..15), add 16 to the index. Hence, "PL_hexdigit[11]" is 'b', and "PL_hexdigit[11+16]" is 'B'. Adding 16 to an index whose representation is '0'..'9' yields the same as not adding 16. Indices outside the range 0..31 result in (bad) undedefined behavior.

"READ_XDIGIT"

Returns the value of an ASCII-range hex digit and advances the string pointer.

Behaviour is only well defined when isXDIGIT(*str) is true.

 U8 READ_XDIGIT(char str*)

"scan_bin"

For backwards compatibility. Use "grok_bin" instead.

NV scan_bin(const char* start, STRLEN len, STRLEN* retlen)

"scan_hex"

For backwards compatibility. Use "grok_hex" instead.

NV scan_hex(const char* start, STRLEN len, STRLEN* retlen)

"scan_oct"

For backwards compatibility. Use "grok_oct" instead.

NV scan_oct(const char* start, STRLEN len, STRLEN* retlen)

"seedDrand01"

This symbol defines the macro to be used in seeding the random number generator (see "Drand01").

void seedDrand01(Rand_seed_t x)

"Strtod"

This is a synonym for "my_strtod".

NV Strtod(NN const char * const s, NULLOK char ** e)

"Strtol"

Platform and configuration independent "strtol". This expands to the appropriate "strotol"-like function based on the platform and Configure options>. For example it could expand to "strtoll" or "strtoq" instead of "strtol".

NV Strtol(NN const char * const s, NULLOK char ** e, int base)

"Strtoul"

Platform and configuration independent "strtoul". This expands to the appropriate "strotoul"-like function based on the platform and Configure options>. For example it could expand to "strtoull" or "strtouq" instead of "strtoul".

NV Strtoul(NN const char * const s, NULLOK char ** e, int base)

Optree construction

"newASSIGNOP"

Constructs, checks, and returns an assignment op. "left" and "right" supply the parameters of the assignment; they are consumed by this function and become part of the constructed op tree.

If "optype" is "OP_ANDASSIGN", "OP_ORASSIGN", or "OP_DORASSIGN", then a suitable conditional optree is constructed. If "optype" is the opcode of a binary operator, such as "OP_BIT_OR", then an op is constructed that performs the binary operation and

assigns the result to the left argument. Either way, if "optype" is non-zero then "flags" has no effect.

If "optype" is zero, then a plain scalar or list assignment is constructed. Which type of assignment it is is automatically determined. "flags" gives the eight bits of "op_flags", except that "OPf_KIDS" will be set automatically, and, shifted up eight bits, the eight bits of "op_private", except that the bit with value 1 or 2 is automatically set as required.

```
OP*  newASSIGNOP(I32 flags, OP* left, I32 optype, OP* right)
```

"newBINOP"

Constructs, checks, and returns an op of any binary type. "type" is the opcode. "flags" gives the eight bits of "op_flags", except that "OPf_KIDS" will be set automatically, and, shifted up eight bits, the eight bits of "op_private", except that the bit with value 1 or 2 is automatically set as required. "first" and "last" supply up to two ops to be the direct children of the binary op; they are consumed by this function and become part of the constructed op tree.

```
OP*  newBINOP(I32 type, I32 flags, OP* first, OP* last)
```

"newCONDOP"

Constructs, checks, and returns a conditional-expression ("cond_expr") op. "flags" gives the eight bits of "op_flags", except that "OPf_KIDS" will be set automatically, and, shifted up eight bits, the eight bits of "op_private", except that the bit with value 1 is automatically set. "first" supplies the expression selecting between the two branches, and "trueop" and "falseop" supply the branches; they are consumed by this function and become part of the constructed op tree.

```
OP*  newCONDOP(I32 flags, OP* first, OP* trueop, OP* falseop)
```

"newDEFSVOP"

Constructs and returns an op to access $_.

```
OP*  newDEFSVOP()
```

"newFOROP"

Constructs, checks, and returns an op tree expressing a "foreach" loop (iteration through a list of values). This is a heavyweight loop, with structure that allows exiting the loop by "last" and suchlike.

"sv" optionally supplies the variable that will be aliased to each item in turn; if null, it defaults to $_. "expr" supplies the list of values to iterate over. "block"

supplies the main body of the loop, and "cont" optionally supplies a "continue" block that operates as a second half of the body. All of these optree inputs are consumed by this function and become part of the constructed op tree.

"flags" gives the eight bits of "op_flags" for the "leaveloop" op and, shifted up eight bits, the eight bits of "op_private" for the "leaveloop" op, except that (in both cases) some bits will be set automatically.

OP* newFOROP(I32 flags, OP* sv, OP* expr, OP* block, OP* cont)

"newGIVENOP"

Constructs, checks, and returns an op tree expressing a "given" block. "cond" supplies the expression to whose value $_ will be locally aliased, and "block" supplies the body of the "given" construct; they are consumed by this function and become part of the constructed op tree. "defsv_off" must be zero (it used to identity the pad slot of lexical $_).

OP* newGIVENOP(OP* cond, OP* block, PADOFFSET defsv_off)

"newGVOP"

Constructs, checks, and returns an op of any type that involves an embedded reference to a GV. "type" is the opcode. "flags" gives the eight bits of "op_flags". "gv" identifies the GV that the op should reference; calling this function does not transfer ownership of any reference to it.

OP* newGVOP(I32 type, I32 flags, GV* gv)

"newLISTOP"

Constructs, checks, and returns an op of any list type. "type" is the opcode. "flags" gives the eight bits of "op_flags", except that "OPf_KIDS" will be set automatically if required. "first" and "last" supply up to two ops to be direct children of the list op; they are consumed by this function and become part of the constructed op tree.

For most list operators, the check function expects all the kid ops to be present already, so calling "newLISTOP(OP_JOIN, ...)" (e.g.) is not appropriate. What you want to do in that case is create an op of type "OP_LIST", append more children to it, and then call "op_convert_list". See "op_convert_list" for more information.

OP* newLISTOP(I32 type, I32 flags, OP* first, OP* last)

"newLOGOP"

Constructs, checks, and returns a logical (flow control) op. "type" is the opcode.

"flags" gives the eight bits of "op_flags", except that "OPf_KIDS" will be set automatically, and, shifted up eight bits, the eight bits of "op_private", except that the bit with value 1 is automatically set. "first" supplies the expression controlling the flow, and "other" supplies the side (alternate) chain of ops; they are consumed by this function and become part of the constructed op tree.

```
OP*  newLOGOP(I32 optype, I32 flags, OP *first, OP *other)
```

"newLOOPEX"

Constructs, checks, and returns a loop-exiting op (such as "goto" or "last"). "type" is the opcode. "label" supplies the parameter determining the target of the op; it is consumed by this function and becomes part of the constructed op tree.

```
OP*  newLOOPEX(I32 type, OP* label)
```

"newLOOPOP"

Constructs, checks, and returns an op tree expressing a loop. This is only a loop in the control flow through the op tree; it does not have the heavyweight loop structure that allows exiting the loop by "last" and suchlike. "flags" gives the eight bits of "op_flags" for the top-level op, except that some bits will be set automatically as required. "expr" supplies the expression controlling loop iteration, and "block" supplies the body of the loop; they are consumed by this function and become part of the constructed op tree. "debuggable" is currently unused and should always be 1.

```
OP*  newLOOPOP(I32 flags, I32 debuggable, OP* expr, OP* block)
```

"newMETHOP"

Constructs, checks, and returns an op of method type with a method name evaluated at runtime. "type" is the opcode. "flags" gives the eight bits of "op_flags", except that "OPf_KIDS" will be set automatically, and, shifted up eight bits, the eight bits of "op_private", except that the bit with value 1 is automatically set. "dynamic_meth" supplies an op which evaluates method name; it is consumed by this function and become part of the constructed op tree. Supported optypes: "OP_METHOD".

```
OP*  newMETHOP(I32 type, I32 flags, OP* dynamic_meth)
```

"newMETHOP_named"

Constructs, checks, and returns an op of method type with a constant method name. "type" is the opcode. "flags" gives the eight bits of "op_flags", and, shifted up eight bits, the eight bits of "op_private". "const_meth" supplies a constant method name; it must be a shared COW string. Supported optypes: "OP_METHOD_NAMED".

OP* newMETHOP_named(I32 type, I32 flags, SV* const_meth)

"newNULLLIST"

Constructs, checks, and returns a new "stub" op, which represents an empty list expression.

OP* newNULLLIST()

"newOP"

Constructs, checks, and returns an op of any base type (any type that has no extra fields). "type" is the opcode. "flags" gives the eight bits of "op_flags", and, shifted up eight bits, the eight bits of "op_private".

OP* newOP(I32 optype, I32 flags)

"newPADOP"

Constructs, checks, and returns an op of any type that involves a reference to a pad element. "type" is the opcode. "flags" gives the eight bits of "op_flags". A pad slot is automatically allocated, and is populated with "sv"; this function takes ownership of one reference to it.

This function only exists if Perl has been compiled to use ithreads.

OP* newPADOP(I32 type, I32 flags, SV* sv)

"newPMOP"

Constructs, checks, and returns an op of any pattern matching type. "type" is the opcode. "flags" gives the eight bits of "op_flags" and, shifted up eight bits, the eight bits of "op_private".

OP* newPMOP(I32 type, I32 flags)

"newPVOP"

Constructs, checks, and returns an op of any type that involves an embedded C-level pointer (PV). "type" is the opcode. "flags" gives the eight bits of "op_flags". "pv" supplies the C-level pointer. Depending on the op type, the memory referenced by "pv" may be freed when the op is destroyed. If the op is of a freeing type, "pv" must have been allocated using "PerlMemShared_malloc".

OP* newPVOP(I32 type, I32 flags, char* pv)

"newRANGE"

Constructs and returns a "range" op, with subordinate "flip" and "flop" ops. "flags" gives the eight bits of "op_flags" for the "flip" op and, shifted up eight bits, the eight bits of "op_private" for both the "flip" and "range" ops, except that the bit

with value 1 is automatically set. "left" and "right" supply the expressions controlling the endpoints of the range; they are consumed by this function and become part of the constructed op tree.

```
OP*  newRANGE(I32 flags, OP* left, OP* right)
```

"newSLICEOP"

Constructs, checks, and returns an "lslice" (list slice) op. "flags" gives the eight bits of "op_flags", except that "OPf_KIDS" will be set automatically, and, shifted up eight bits, the eight bits of "op_private", except that the bit with value 1 or 2 is automatically set as required. "listval" and "subscript" supply the parameters of the slice; they are consumed by this function and become part of the constructed op tree.

```
OP*  newSLICEOP(I32 flags, OP* subscript, OP* listop)
```

"newSTATEOP"

Constructs a state op (COP). The state op is normally a "nextstate" op, but will be a "dbstate" op if debugging is enabled for currently-compiled code. The state op is populated from "PL_curcop" (or "PL_compiling"). If "label" is non-null, it supplies the name of a label to attach to the state op; this function takes ownership of the memory pointed at by "label", and will free it. "flags" gives the eight bits of "op_flags" for the state op.

If "o" is null, the state op is returned. Otherwise the state op is combined with "o" into a "lineseq" list op, which is returned. "o" is consumed by this function and becomes part of the returned op tree.

```
OP*  newSTATEOP(I32 flags, char* label, OP* o)
```

"newSVOP"

Constructs, checks, and returns an op of any type that involves an embedded SV. "type" is the opcode. "flags" gives the eight bits of "op_flags". "sv" gives the SV to embed in the op; this function takes ownership of one reference to it.

```
OP*  newSVOP(I32 type, I32 flags, SV* sv)
```

"newTRYCATCHOP"

NOTE: "newTRYCATCHOP" is experimental and may change or be removed without notice.

Constructs and returns a conditional execution statement that implements the "try"/"catch" semantics. First the op tree in "tryblock" is executed, inside a context that traps exceptions. If an exception occurs then the optree in "catchblock" is executed, with the trapped exception set into the lexical variable given by

"catchvar" (which must be an op of type "OP_PADSV"). All the optrees are consumed by this function and become part of the returned op tree.

The "flags" argument is currently ignored.

```
OP*  newTRYCATCHOP(I32 flags, OP* tryblock, OP *catchvar,
                   OP* catchblock)
```

"newUNOP"

Constructs, checks, and returns an op of any unary type. "type" is the opcode. "flags" gives the eight bits of "op_flags", except that "OPf_KIDS" will be set automatically if required, and, shifted up eight bits, the eight bits of "op_private", except that the bit with value 1 is automatically set. "first" supplies an optional op to be the direct child of the unary op; it is consumed by this function and become part of the constructed op tree.

```
OP*  newUNOP(I32 type, I32 flags, OP* first)
```

"newUNOP_AUX"

Similar to "newUNOP", but creates an "UNOP_AUX" struct instead, with "op_aux" initialised to "aux"

```
OP*  newUNOP_AUX(I32 type, I32 flags, OP* first,
                 UNOP_AUX_item *aux)
```

"newWHENOP"

Constructs, checks, and returns an op tree expressing a "when" block. "cond" supplies the test expression, and "block" supplies the block that will be executed if the test evaluates to true; they are consumed by this function and become part of the constructed op tree. "cond" will be interpreted DWIMically, often as a comparison against $_, and may be null to generate a "default" block.

```
OP*  newWHENOP(OP* cond, OP* block)
```

"newWHILEOP"

Constructs, checks, and returns an op tree expressing a "while" loop. This is a heavyweight loop, with structure that allows exiting the loop by "last" and suchlike. "loop" is an optional preconstructed "enterloop" op to use in the loop; if it is null then a suitable op will be constructed automatically. "expr" supplies the loop's controlling expression. "block" supplies the main body of the loop, and "cont" optionally supplies a "continue" block that operates as a second half of the body. All of these optree inputs are consumed by this function and become part of the

constructed op tree.

"flags" gives the eight bits of "op_flags" for the "leaveloop" op and, shifted up eight bits, the eight bits of "op_private" for the "leaveloop" op, except that (in both cases) some bits will be set automatically. "debuggable" is currently unused and should always be 1. "has_my" can be supplied as true to force the loop body to be enclosed in its own scope.

```
OP*  newWHILEOP(I32 flags, I32 debuggable, LOOP* loop, OP* expr,
                OP* block, OP* cont, I32 has_my)
```

"PL_opfreehook"

When non-"NULL", the function pointed by this variable will be called each time an OP is freed with the corresponding OP as the argument. This allows extensions to free any extra attribute they have locally attached to an OP. It is also assured to first fire for the parent OP and then for its kids.

When you replace this variable, it is considered a good practice to store the possibly previously installed hook and that you recall it inside your own.

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

```
Perl_ophook_t  PL_opfreehook
```

"PL_peepp"

Pointer to the per-subroutine peephole optimiser. This is a function that gets called at the end of compilation of a Perl subroutine (or equivalently independent piece of Perl code) to perform fixups of some ops and to perform small-scale optimisations. The function is called once for each subroutine that is compiled, and is passed, as sole parameter, a pointer to the op that is the entry point to the subroutine. It modifies the op tree in place.

The peephole optimiser should never be completely replaced. Rather, add code to it by wrapping the existing optimiser. The basic way to do this can be seen in "Compile pass 3: peephole optimization" in perlguts. If the new code wishes to operate on ops throughout the subroutine's structure, rather than just at the top level, it is likely to be more convenient to wrap the "PL_rpeepp" hook.

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

```
peep_t  PL_peepp
```

"PL_rpeepp"

Pointer to the recursive peephole optimiser. This is a function that gets called at the end of compilation of a Perl subroutine (or equivalently independent piece of Perl code) to perform fixups of some ops and to perform small-scale optimisations. The function is called once for each chain of ops linked through their "op_next" fields; it is recursively called to handle each side chain. It is passed, as sole parameter, a pointer to the op that is at the head of the chain. It modifies the op tree in place.

The peephole optimiser should never be completely replaced. Rather, add code to it by wrapping the existing optimiser. The basic way to do this can be seen in "Compile pass 3: peephole optimization" in perlguts. If the new code wishes to operate only on ops at a subroutine's top level, rather than throughout the structure, it is likely to be more convenient to wrap the "PL_peepp" hook.

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

```
peep_t  PL_rpeepp
```

Optree Manipulation Functions

"alloccopstash"

NOTE: "alloccopstash" is experimental and may change or be removed without notice.

Available only under threaded builds, this function allocates an entry in "PL_stashpad" for the stash passed to it.

```
PADOFFSET  alloccopstash(HV *hv)
```

"block_end"

Handles compile-time scope exit. "floor" is the savestack index returned by "block_start", and "seq" is the body of the block. Returns the block, possibly modified.

```
OP*  block_end(I32 floor, OP* seq)
```

"block_start"

Handles compile-time scope entry. Arranges for hints to be restored on block exit and also handles pad sequence numbers to make lexical variables scope right. Returns a savestack index for use with "block_end".

```
int  block_start(int full)
```

"ck_entersub_args_list"

Performs the default fixup of the arguments part of an "entersub" op tree. This consists of applying list context to each of the argument ops. This is the standard treatment used on a call marked with "&", or a method call, or a call through a subroutine reference, or any other call where the callee can't be identified at compile time, or a call where the callee has no prototype.

```
OP*  ck_entersub_args_list(OP *entersubop)
```

"ck_entersub_args_proto"

Performs the fixup of the arguments part of an "entersub" op tree based on a subroutine prototype. This makes various modifications to the argument ops, from applying context up to inserting "refgen" ops, and checking the number and syntactic types of arguments, as directed by the prototype. This is the standard treatment used on a subroutine call, not marked with "&", where the callee can be identified at compile time and has a prototype.

"protosv" supplies the subroutine prototype to be applied to the call. It may be a normal defined scalar, of which the string value will be used. Alternatively, for convenience, it may be a subroutine object (a "CV*" that has been cast to "SV*") which has a prototype. The prototype supplied, in whichever form, does not need to match the actual callee referenced by the op tree.

If the argument ops disagree with the prototype, for example by having an unacceptable number of arguments, a valid op tree is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. In the error message, the callee is referred to by the name defined by the "namegv" parameter.

```
OP*  ck_entersub_args_proto(OP *entersubop, GV *namegv,
                            SV *protosv)
```

"ck_entersub_args_proto_or_list"

Performs the fixup of the arguments part of an "entersub" op tree either based on a subroutine prototype or using default list-context processing. This is the standard treatment used on a subroutine call, not marked with "&", where the callee can be identified at compile time.

"protosv" supplies the subroutine prototype to be applied to the call, or indicates that there is no prototype. It may be a normal scalar, in which case if it is defined then the string value will be used as a prototype, and if it is undefined then there

is no prototype. Alternatively, for convenience, it may be a subroutine object (a "CV*" that has been cast to "SV*"), of which the prototype will be used if it has one. The prototype (or lack thereof) supplied, in whichever form, does not need to match the actual callee referenced by the op tree.

If the argument ops disagree with the prototype, for example by having an unacceptable number of arguments, a valid op tree is returned anyway. The error is reflected in the parser state, normally resulting in a single exception at the top level of parsing which covers all the compilation errors that occurred. In the error message, the callee is referred to by the name defined by the "namegv" parameter.

```
OP*  ck_entersub_args_proto_or_list(OP *entersubop, GV *namegv,
                                    SV *protosv)
```

"cv_const_sv"

If "cv" is a constant sub eligible for inlining, returns the constant value returned by the sub. Otherwise, returns "NULL".

Constant subs can be created with "newCONSTSUB" or as described in "Constant Functions" in perlsub.

```
SV*  cv_const_sv(const CV *const cv)
```

"cv_get_call_checker"

The original form of "cv_get_call_checker_flags", which does not return checker flags. When using a checker function returned by this function, it is only safe to call it with a genuine GV as its "namegv" argument.

```
void  cv_get_call_checker(CV *cv, Perl_call_checker *ckfun_p,
                          SV **ckobj_p)
```

"cv_get_call_checker_flags"

Retrieves the function that will be used to fix up a call to "cv". Specifically, the function is applied to an "entersub" op tree for a subroutine call, not marked with "&", where the callee can be identified at compile time as "cv".

The C-level function pointer is returned in *ckfun_p, an SV argument for it is returned in *ckobj_p, and control flags are returned in *ckflags_p. The function is intended to be called in this manner:

```
entersubop = (*ckfun_p)(aTHX_ entersubop, namegv, (*ckobj_p));
```

In this call, "entersubop" is a pointer to the "entersub" op, which may be replaced by the check function, and "namegv" supplies the name that should be used by the check

function to refer to the callee of the "entersub" op if it needs to emit any diagnostics. It is permitted to apply the check function in non-standard situations, such as to a call to a different subroutine or to a method call.

"namegv" may not actually be a GV. If the "CALL_CHECKER_REQUIRE_GV" bit is clear in *ckflags_p, it is permitted to pass a CV or other SV instead, anything that can be used as the first argument to "cv_name". If the "CALL_CHECKER_REQUIRE_GV" bit is set in *ckflags_p then the check function requires "namegv" to be a genuine GV.

By default, the check function is Perl_ck_entersub_args_proto_or_list, the SV parameter is "cv" itself, and the "CALL_CHECKER_REQUIRE_GV" flag is clear. This implements standard prototype processing. It can be changed, for a particular subroutine, by "cv_set_call_checker_flags".

If the "CALL_CHECKER_REQUIRE_GV" bit is set in "gflags" then it indicates that the caller only knows about the genuine GV version of "namegv", and accordingly the corresponding bit will always be set in *ckflags_p, regardless of the check function's recorded requirements. If the "CALL_CHECKER_REQUIRE_GV" bit is clear in "gflags" then it indicates the caller knows about the possibility of passing something other than a GV as "namegv", and accordingly the corresponding bit may be either set or clear in *ckflags_p, indicating the check function's recorded requirements.

"gflags" is a bitset passed into "cv_get_call_checker_flags", in which only the "CALL_CHECKER_REQUIRE_GV" bit currently has a defined meaning (for which see above). All other bits should be clear.

```
 void  cv_get_call_checker_flags(CV *cv, U32 gflags,
                                 Perl_call_checker *ckfun_p,
                                 SV **ckobj_p, U32 *ckflags_p)
```

"cv_set_call_checker"

The original form of "cv_set_call_checker_flags", which passes it the "CALL_CHECKER_REQUIRE_GV" flag for backward-compatibility. The effect of that flag setting is that the check function is guaranteed to get a genuine GV as its "namegv" argument.

```
 void  cv_set_call_checker(CV *cv, Perl_call_checker ckfun,
                           SV *ckobj)
```

"cv_set_call_checker_flags"

Sets the function that will be used to fix up a call to "cv". Specifically, the

function is applied to an "entersub" op tree for a subroutine call, not marked with "&", where the callee can be identified at compile time as "cv".

The C-level function pointer is supplied in "ckfun", an SV argument for it is supplied in "ckobj", and control flags are supplied in "ckflags". The function should be defined like this:

```
    STATIC OP * ckfun(pTHX_ OP *op, GV *namegv, SV *ckobj)
```

It is intended to be called in this manner:

```
    entersubop = ckfun(aTHX_ entersubop, namegv, ckobj);
```

In this call, "entersubop" is a pointer to the "entersub" op, which may be replaced by the check function, and "namegv" supplies the name that should be used by the check function to refer to the callee of the "entersub" op if it needs to emit any diagnostics. It is permitted to apply the check function in non-standard situations, such as to a call to a different subroutine or to a method call.

"namegv" may not actually be a GV. For efficiency, perl may pass a CV or other SV instead. Whatever is passed can be used as the first argument to "cv_name". You can force perl to pass a GV by including "CALL_CHECKER_REQUIRE_GV" in the "ckflags".

"ckflags" is a bitset, in which only the "CALL_CHECKER_REQUIRE_GV" bit currently has a defined meaning (for which see above). All other bits should be clear.

The current setting for a particular CV can be retrieved by "cv_get_call_checker_flags".

```
 void  cv_set_call_checker_flags(CV *cv, Perl_call_checker ckfun,
                                 SV *ckobj, U32 ckflags)
```

"LINKLIST"

Given the root of an optree, link the tree in execution order using the "op_next" pointers and return the first op executed. If this has already been done, it will not be redone, and "o->op_next" will be returned. If "o->op_next" is not already set, "o" should be at least an "UNOP".

```
 OP*  LINKLIST(OP *o)
```

"newATTRSUB"

Construct a Perl subroutine, also performing some surrounding jobs.

This is the same as ""newATTRSUB_x"" in perlintern with its "o_is_gv" parameter set to FALSE. This means that if "o" is null, the new sub will be anonymous; otherwise the name will be derived from "o" in the way described (as with all other details) in

""newATTRSUB_x"" in perlintern.

CV* newATTRSUB(I32 floor, OP *o, OP *proto, OP *attrs, OP *block)

"newCONSTSUB"

Behaves like "newCONSTSUB_flags", except that "name" is nul-terminated rather than of counted length, and no flags are set. (This means that "name" is always interpreted as Latin-1.)

CV* newCONSTSUB(HV* stash, const char* name, SV* sv)

"newCONSTSUB_flags"

Construct a constant subroutine, also performing some surrounding jobs. A scalar constant-valued subroutine is eligible for inlining at compile-time, and in Perl code can be created by "sub?FOO?()?{?123?}". Other kinds of constant subroutine have other treatment.

The subroutine will have an empty prototype and will ignore any arguments when called. Its constant behaviour is determined by "sv". If "sv" is null, the subroutine will yield an empty list. If "sv" points to a scalar, the subroutine will always yield that scalar. If "sv" points to an array, the subroutine will always yield a list of the elements of that array in list context, or the number of elements in the array in scalar context. This function takes ownership of one counted reference to the scalar or array, and will arrange for the object to live as long as the subroutine does. If "sv" points to a scalar then the inlining assumes that the value of the scalar will never change, so the caller must ensure that the scalar is not subsequently written to. If "sv" points to an array then no such assumption is made, so it is ostensibly safe to mutate the array or its elements, but whether this is really supported has not been determined.

The subroutine will have "CvFILE" set according to "PL_curcop". Other aspects of the subroutine will be left in their default state. The caller is free to mutate the subroutine beyond its initial state after this function has returned.

If "name" is null then the subroutine will be anonymous, with its "CvGV" referring to an "__ANON__" glob. If "name" is non-null then the subroutine will be named accordingly, referenced by the appropriate glob. "name" is a string of length "len" bytes giving a sigilless symbol name, in UTF-8 if "flags" has the "SVf_UTF8" bit set and in Latin-1 otherwise. The name may be either qualified or unqualified. If the name is unqualified then it defaults to being in the stash specified by "stash" if

that is non-null, or to "PL_curstash" if "stash" is null. The symbol is always added to the stash if necessary, with "GV_ADDMULTI" semantics.

"flags" should not have bits set other than "SVf_UTF8".

If there is already a subroutine of the specified name, then the new sub will replace the existing one in the glob. A warning may be generated about the redefinition.

If the subroutine has one of a few special names, such as "BEGIN" or "END", then it will be claimed by the appropriate queue for automatic running of phase-related subroutines. In this case the relevant glob will be left not containing any subroutine, even if it did contain one before. Execution of the subroutine will likely be a no-op, unless "sv" was a tied array or the caller modified the subroutine in some interesting way before it was executed. In the case of "BEGIN", the treatment is buggy: the sub will be executed when only half built, and may be deleted prematurely, possibly causing a crash.

The function returns a pointer to the constructed subroutine. If the sub is anonymous then ownership of one counted reference to the subroutine is transferred to the caller. If the sub is named then the caller does not get ownership of a reference. In most such cases, where the sub has a non-phase name, the sub will be alive at the point it is returned by virtue of being contained in the glob that names it. A phase-named subroutine will usually be alive by virtue of the reference owned by the phase's automatic run queue. A "BEGIN" subroutine may have been destroyed already by the time this function returns, but currently bugs occur in that case before the caller gets control. It is the caller's responsibility to ensure that it knows which of these situations applies.

```
 CV*  newCONSTSUB_flags(HV* stash, const char* name, STRLEN len,
                        U32 flags, SV* sv)
```

"newSUB"

Like "newATTRSUB", but without attributes.

```
 CV*  newSUB(I32 floor, OP* o, OP* proto, OP* block)
```

"newXS"

Used by "xsubpp" to hook up XSUBs as Perl subs. "filename" needs to be static storage, as it is used directly as CvFILE(), without a copy being made.

"op_append_elem"

Append an item to the list of ops contained directly within a list-type op, returning

the lengthened list. "first" is the list-type op, and "last" is the op to append to the list. "optype" specifies the intended opcode for the list. If "first" is not already a list of the right type, it will be upgraded into one. If either "first" or "last" is null, the other is returned unchanged.

```
 OP*  op_append_elem(I32 optype, OP* first, OP* last)
```

"op_append_list"

Concatenate the lists of ops contained directly within two list-type ops, returning the combined list. "first" and "last" are the list-type ops to concatenate. "optype" specifies the intended opcode for the list. If either "first" or "last" is not already a list of the right type, it will be upgraded into one. If either "first" or "last" is null, the other is returned unchanged.

```
 OP*  op_append_list(I32 optype, OP* first, OP* last)
```

"OP_CLASS"

Return the class of the provided OP: that is, which of the *OP structures it uses. For core ops this currently gets the information out of "PL_opargs", which does not always accurately reflect the type used; in v5.26 onwards, see also the function "op_class" which can do a better job of determining the used type.

For custom ops the type is returned from the registration, and it is up to the registree to ensure it is accurate. The value returned will be one of the "OA_"* constants from op.h.

```
 U32  OP_CLASS(OP *o)
```

"op_contextualize"

Applies a syntactic context to an op tree representing an expression. "o" is the op tree, and "context" must be "G_SCALAR", "G_ARRAY", or "G_VOID" to specify the context to apply. The modified op tree is returned.

```
 OP*  op_contextualize(OP* o, I32 context)
```

"op_convert_list"

Converts "o" into a list op if it is not one already, and then converts it into the specified "type", calling its check function, allocating a target if it needs one, and folding constants.

A list-type op is usually constructed one kid at a time via "newLISTOP", "op_prepend_elem" and "op_append_elem". Then finally it is passed to "op_convert_list" to make it the right type.

OP* op_convert_list(I32 optype, I32 flags, OP* o)

"OP_DESC"

Return a short description of the provided OP.

const char * OP_DESC(OP *o)

"op_free"

Free an op and its children. Only use this when an op is no longer linked to from any optree.

void op_free(OP* arg)

"OpHAS_SIBLING"

Returns true if "o" has a sibling

bool OpHAS_SIBLING(OP *o)

"OpLASTSIB_set"

Marks "o" as having no further siblings and marks o as having the specified parent. See also "OpMORESIB_set" and "OpMAYBESIB_set". For a higher-level interface, see "op_sibling_splice".

void OpLASTSIB_set(OP *o, OP *parent)

"op_linklist"

This function is the implementation of the "LINKLIST" macro. It should not be called directly.

OP* op_linklist(OP *o)

"op_lvalue"

NOTE: "op_lvalue" is experimental and may change or be removed without notice.

Propagate lvalue ("modifiable") context to an op and its children. "type" represents the context type, roughly based on the type of op that would do the modifying, although "local()" is represented by "OP_NULL", because it has no op type of its own (it is signalled by a flag on the lvalue op).

This function detects things that can't be modified, such as "$x+1", and generates errors for them. For example, "$x+1 = 2" would cause it to be called with an op of type "OP_ADD" and a "type" argument of "OP_SASSIGN".

It also flags things that need to behave specially in an lvalue context, such as "$$x = 5" which might have to vivify a reference in $x.

OP* op_lvalue(OP* o, I32 type)

"OpMAYBESIB_set"

Conditionally does "OpMORESIB_set" or "OpLASTSIB_set" depending on whether "sib" is non-null. For a higher-level interface, see "op_sibling_splice".

```
void  OpMAYBESIB_set(OP *o, OP *sib, OP *parent)
```

"OpMORESIB_set"

Sets the sibling of "o" to the non-zero value "sib". See also "OpLASTSIB_set" and "OpMAYBESIB_set". For a higher-level interface, see "op_sibling_splice".

```
void  OpMORESIB_set(OP *o, OP *sib)
```

"OP_NAME"

Return the name of the provided OP. For core ops this looks up the name from the op_type; for custom ops from the op_ppaddr.

```
const char *  OP_NAME(OP *o)
```

"op_null"

Neutralizes an op when it is no longer needed, but is still linked to from other ops.

```
void  op_null(OP* o)
```

"op_parent"

Returns the parent OP of "o", if it has a parent. Returns "NULL" otherwise.

```
OP*  op_parent(OP *o)
```

"op_prepend_elem"

Prepend an item to the list of ops contained directly within a list-type op, returning the lengthened list. "first" is the op to prepend to the list, and "last" is the list-type op. "optype" specifies the intended opcode for the list. If "last" is not already a list of the right type, it will be upgraded into one. If either "first" or "last" is null, the other is returned unchanged.

```
OP*  op_prepend_elem(I32 optype, OP* first, OP* last)
```

"op_scope"

NOTE: "op_scope" is experimental and may change or be removed without notice.

Wraps up an op tree with some additional ops so that at runtime a dynamic scope will be created. The original ops run in the new dynamic scope, and then, provided that they exit normally, the scope will be unwound. The additional ops used to create and unwind the dynamic scope will normally be an "enter"/"leave" pair, but a "scope" op may be used instead if the ops are simple enough to not need the full dynamic scope structure.

```
OP*  op_scope(OP* o)
```

"OpSIBLING"

Returns the sibling of "o", or "NULL" if there is no sibling

OP* OpSIBLING(OP *o)

"op_sibling_splice"

A general function for editing the structure of an existing chain of op_sibling nodes.

By analogy with the perl-level "splice()" function, allows you to delete zero or more sequential nodes, replacing them with zero or more different nodes. Performs the necessary op_first/op_last housekeeping on the parent node and op_sibling manipulation on the children. The last deleted node will be marked as the last node by updating the op_sibling/op_sibparent or op_moresib field as appropriate.

Note that op_next is not manipulated, and nodes are not freed; that is the responsibility of the caller. It also won't create a new list op for an empty list etc; use higher-level functions like op_append_elem() for that.

"parent" is the parent node of the sibling chain. It may passed as "NULL" if the splicing doesn't affect the first or last op in the chain.

"start" is the node preceding the first node to be spliced. Node(s) following it will be deleted, and ops will be inserted after it. If it is "NULL", the first node onwards is deleted, and nodes are inserted at the beginning.

"del_count" is the number of nodes to delete. If zero, no nodes are deleted. If -1 or greater than or equal to the number of remaining kids, all remaining kids are deleted.

"insert" is the first of a chain of nodes to be inserted in place of the nodes. If "NULL", no nodes are inserted.

The head of the chain of deleted ops is returned, or "NULL" if no ops were deleted.

For example:

action	before	after	returns
	P A-B-C-D	P A-X-Y-Z-D	
splice(P, A, 2, X-Y-Z)	|	|	B-C
	P A-B-C-D	P X-Y-B-C-D	
splice(P, NULL, 1, X-Y)	|	|	A

```
                        P         P
    splice(P, NULL, 3, NULL)  |         |          A-B-C
                        A-B-C-D   D
                        P         P
    splice(P, B, 0, X-Y)      |         |          NULL
                        A-B-C-D   A-B-X-Y-C-D
```

For lower-level direct manipulation of "op_sibparent" and "op_moresib", see "OpMORESIB_set", "OpLASTSIB_set", "OpMAYBESIB_set".

```
 OP*  op_sibling_splice(OP *parent, OP *start, int del_count,
                        OP* insert)
```

"OP_TYPE_IS"

Returns true if the given OP is not a "NULL" pointer and if it is of the given type.

The negation of this macro, "OP_TYPE_ISNT" is also available as well as "OP_TYPE_IS_NN" and "OP_TYPE_ISNT_NN" which elide the NULL pointer check.

```
 bool  OP_TYPE_IS(OP *o, Optype type)
```

"OP_TYPE_IS_OR_WAS"

Returns true if the given OP is not a NULL pointer and if it is of the given type or used to be before being replaced by an OP of type OP_NULL.

The negation of this macro, "OP_TYPE_ISNT_AND_WASNT" is also available as well as "OP_TYPE_IS_OR_WAS_NN" and "OP_TYPE_ISNT_AND_WASNT_NN" which elide the "NULL" pointer check.

```
 bool  OP_TYPE_IS_OR_WAS(OP *o, Optype type)
```

"rv2cv_op_cv"

Examines an op, which is expected to identify a subroutine at runtime, and attempts to determine at compile time which subroutine it identifies. This is normally used during Perl compilation to determine whether a prototype can be applied to a function call. "cvop" is the op being considered, normally an "rv2cv" op. A pointer to the identified subroutine is returned, if it could be determined statically, and a null pointer is returned if it was not possible to determine statically.

Currently, the subroutine can be identified statically if the RV that the "rv2cv" is to operate on is provided by a suitable "gv" or "const" op. A "gv" op is suitable if the GV's CV slot is populated. A "const" op is suitable if the constant value must be an RV pointing to a CV. Details of this process may change in future versions of

Perl. If the "rv2cv" op has the "OPpENTERSUB_AMPER" flag set then no attempt is made to identify the subroutine statically: this flag is used to suppress compile-time magic on a subroutine call, forcing it to use default runtime behaviour.

If "flags" has the bit "RV2CVOPCV_MARK_EARLY" set, then the handling of a GV reference is modified. If a GV was examined and its CV slot was found to be empty, then the "gv" op has the "OPpEARLY_CV" flag set. If the op is not optimised away, and the CV slot is later populated with a subroutine having a prototype, that flag eventually triggers the warning "called too early to check prototype".

If "flags" has the bit "RV2CVOPCV_RETURN_NAME_GV" set, then instead of returning a pointer to the subroutine it returns a pointer to the GV giving the most appropriate name for the subroutine in this context. Normally this is just the "CvGV" of the subroutine, but for an anonymous ("CvANON") subroutine that is referenced through a GV it will be the referencing GV. The resulting "GV*" is cast to "CV*" to be returned.

A null pointer is returned as usual if there is no statically-determinable subroutine.

```
CV*  rv2cv_op_cv(OP *cvop, U32 flags)
```

Pack and Unpack

"pack_cat"

"DEPRECATED!" It is planned to remove "pack_cat" from a future release of Perl. Do not use it for new code; remove it from existing code.

The engine implementing "pack()" Perl function. Note: parameters "next_in_list" and "flags" are not used. This call should not be used; use "packlist" instead.

```
void  pack_cat(SV *cat, const char *pat, const char *patend,
               SV **beglist, SV **endlist, SV ***next_in_list,
               U32 flags)
```

"packlist"

The engine implementing "pack()" Perl function.

```
void  packlist(SV *cat, const char *pat, const char *patend,
               SV **beglist, SV **endlist)
```

"unpack_str"

"DEPRECATED!" It is planned to remove "unpack_str" from a future release of Perl. Do not use it for new code; remove it from existing code.

The engine implementing "unpack()" Perl function. Note: parameters "strbeg", "new_s" and "ocnt" are not used. This call should not be used, use "unpackstring" instead.

SSize_t unpack_str(const char *pat, const char *patend,
const char *s, const char *strbeg,
const char *strend, char **new_s, I32 ocnt,
U32 flags)

"unpackstring"

The engine implementing the "unpack()" Perl function.

Using the template "pat..patend", this function unpacks the string "s..strend" into a number of mortal SVs, which it pushes onto the perl argument (@_) stack (so you will need to issue a "PUTBACK" before and "SPAGAIN" after the call to this function). It returns the number of pushed elements.

The "strend" and "patend" pointers should point to the byte following the last character of each string.

Although this function returns its values on the perl argument stack, it doesn't take any parameters from that stack (and thus in particular there's no need to do a "PUSHMARK" before calling it, unlike "call_pv" for example).

SSize_t unpackstring(const char *pat, const char *patend,
const char *s, const char *strend,
U32 flags)

Pad Data Structures

"CvPADLIST"

NOTE: "CvPADLIST" is experimental and may change or be removed without notice.

CV's can have CvPADLIST(cv) set to point to a PADLIST. This is the CV's scratchpad, which stores lexical variables and opcode temporary and per-thread values.

For these purposes "formats" are a kind-of CV; eval""s are too (except they're not callable at will and are always thrown away after the eval"" is done executing).

Require'd files are simply evals without any outer lexical scope.

XSUBs do not have a "CvPADLIST". "dXSTARG" fetches values from "PL_curpad", but that is really the callers pad (a slot of which is allocated by every entersub). Do not get or set "CvPADLIST" if a CV is an XSUB (as determined by "CvISXSUB()"), "CvPADLIST" slot is reused for a different internal purpose in XSUBs.

The PADLIST has a C array where pads are stored.

The 0th entry of the PADLIST is a PADNAMELIST which represents the "names" or rather the "static type information" for lexicals. The individual elements of a PADNAMELIST

are PADNAMEs. Future refactorings might stop the PADNAMELIST from being stored in the PADLIST's array, so don't rely on it. See "PadlistNAMES".

The CvDEPTH'th entry of a PADLIST is a PAD (an AV) which is the stack frame at that depth of recursion into the CV. The 0th slot of a frame AV is an AV which is @_.

Other entries are storage for variables and op targets.

Iterating over the PADNAMELIST iterates over all possible pad items. Pad slots for targets ("SVs_PADTMP") and GVs end up having &PL_padname_undef "names", while slots for constants have &PL_padname_const "names" (see "pad_alloc"). That &PL_padname_undef and &PL_padname_const are used is an implementation detail subject to change. To test for them, use "!PadnamePV(name)" and "PadnamePV(name)?&&?!PadnameLEN(name)", respectively.

Only "my"/"our" variable slots get valid names. The rest are op targets/GVs/constants which are statically allocated or resolved at compile time. These don't have names by which they can be looked up from Perl code at run time through eval"" the way "my"/"our" variables can be. Since they can't be looked up by "name" but only by their index allocated at compile time (which is usually in "PL_op->op_targ"), wasting a name SV for them doesn't make sense.

The pad names in the PADNAMELIST have their PV holding the name of the variable. The "COP_SEQ_RANGE_LOW" and "_HIGH" fields form a range (low+1..high inclusive) of cop_seq numbers for which the name is valid. During compilation, these fields may hold the special value PERL_PADSEQ_INTRO to indicate various stages:

```
 COP_SEQ_RANGE_LOW        _HIGH
 -----------------        -----
 PERL_PADSEQ_INTRO            0   variable not yet introduced:
                                  { my ($x
 valid-seq#   PERL_PADSEQ_INTRO   variable in scope:
                                  { my ($x);
 valid-seq#          valid-seq#   compilation of scope complete:
                                  { my ($x); .... }
```

When a lexical var hasn't yet been introduced, it already exists from the perspective of duplicate declarations, but not for variable lookups, e.g.

```
    my ($x, $x); # '"my" variable $x masks earlier declaration'
    my $x = $x;  # equal to my $x = $::x;
```

For typed lexicals "PadnameTYPE" points at the type stash. For "our" lexicals, "PadnameOURSTASH" points at the stash of the associated global (so that duplicate "our" declarations in the same package can be detected). "PadnameGEN" is sometimes used to store the generation number during compilation.

If "PadnameOUTER" is set on the pad name, then that slot in the frame AV is a REFCNT'ed reference to a lexical from "outside". Such entries are sometimes referred to as 'fake'. In this case, the name does not use 'low' and 'high' to store a cop_seq range, since it is in scope throughout. Instead 'high' stores some flags containing info about the real lexical (is it declared in an anon, and is it capable of being instantiated multiple times?), and for fake ANONs, 'low' contains the index within the parent's pad where the lexical's value is stored, to make cloning quicker.

If the 'name' is "&" the corresponding entry in the PAD is a CV representing a possible closure.

Note that formats are treated as anon subs, and are cloned each time write is called (if necessary).

The flag "SVs_PADSTALE" is cleared on lexicals each time the "my()" is executed, and set on scope exit. This allows the "Variable $x is not available" warning to be generated in evals, such as

```
{ my $x = 1; sub f { eval '$x'} } f();
```

For state vars, "SVs_PADSTALE" is overloaded to mean 'not yet initialised', but this internal state is stored in a separate pad entry.

```
PADLIST *  CvPADLIST(CV *cv)
```

"pad_add_name_pvs"

Exactly like "pad_add_name_pvn", but takes a literal string instead of a string/length pair.

```
PADOFFSET  pad_add_name_pvs("name", U32 flags, HV *typestash,
                            HV *ourstash)
```

"PadARRAY"

NOTE: "PadARRAY" is experimental and may change or be removed without notice.

The C array of pad entries.

```
SV **  PadARRAY(PAD * pad)
```

"pad_findmy_pvs"

Exactly like "pad_findmy_pvn", but takes a literal string instead of a string/length

pair.

PADOFFSET pad_findmy_pvs("name", U32 flags)

"PadlistARRAY"

NOTE: "PadlistARRAY" is experimental and may change or be removed without notice.

The C array of a padlist, containing the pads. Only subscript it with numbers >= 1,

as the 0th entry is not guaranteed to remain usable.

PAD ** PadlistARRAY(PADLIST * padlist)

"PadlistMAX"

NOTE: "PadlistMAX" is experimental and may change or be removed without notice.

The index of the last allocated space in the padlist. Note that the last pad may be

in an earlier slot. Any entries following it will be "NULL" in that case.

SSize_t PadlistMAX(PADLIST * padlist)

"PadlistNAMES"

NOTE: "PadlistNAMES" is experimental and may change or be removed without notice.

The names associated with pad entries.

PADNAMELIST * PadlistNAMES(PADLIST * padlist)

"PadlistNAMESARRAY"

NOTE: "PadlistNAMESARRAY" is experimental and may change or be removed without notice.

The C array of pad names.

PADNAME ** PadlistNAMESARRAY(PADLIST * padlist)

"PadlistNAMESMAX"

NOTE: "PadlistNAMESMAX" is experimental and may change or be removed without notice.

The index of the last pad name.

SSize_t PadlistNAMESMAX(PADLIST * padlist)

"PadlistREFCNT"

NOTE: "PadlistREFCNT" is experimental and may change or be removed without notice.

The reference count of the padlist. Currently this is always 1.

U32 PadlistREFCNT(PADLIST * padlist)

"PadMAX"

NOTE: "PadMAX" is experimental and may change or be removed without notice.

The index of the last pad entry.

SSize_t PadMAX(PAD * pad)

"PadnameLEN"

NOTE: "PadnameLEN" is experimental and may change or be removed without notice.

The length of the name.

```
STRLEN  PadnameLEN(PADNAME * pn)
```

"PadnamelistARRAY"

NOTE: "PadnamelistARRAY" is experimental and may change or be removed without notice.

The C array of pad names.

```
PADNAME **  PadnamelistARRAY(PADNAMELIST * pnl)
```

"PadnamelistMAX"

NOTE: "PadnamelistMAX" is experimental and may change or be removed without notice.

The index of the last pad name.

```
SSize_t  PadnamelistMAX(PADNAMELIST * pnl)
```

"PadnamelistREFCNT"

NOTE: "PadnamelistREFCNT" is experimental and may change or be removed without notice.

The reference count of the pad name list.

```
SSize_t  PadnamelistREFCNT(PADNAMELIST * pnl)
```

"PadnamelistREFCNT_dec"

NOTE: "PadnamelistREFCNT_dec" is experimental and may change or be removed without notice.

Lowers the reference count of the pad name list.

```
void  PadnamelistREFCNT_dec(PADNAMELIST * pnl)
```

"PadnamePV"

NOTE: "PadnamePV" is experimental and may change or be removed without notice.

The name stored in the pad name struct. This returns "NULL" for a target slot.

```
char *  PadnamePV(PADNAME * pn)
```

"PadnameREFCNT"

NOTE: "PadnameREFCNT" is experimental and may change or be removed without notice.

The reference count of the pad name.

```
SSize_t  PadnameREFCNT(PADNAME * pn)
```

"PadnameREFCNT_dec"

NOTE: "PadnameREFCNT_dec" is experimental and may change or be removed without notice.

Lowers the reference count of the pad name.

```
void  PadnameREFCNT_dec(PADNAME * pn)
```

"PadnameSV"

NOTE: "PadnameSV" is experimental and may change or be removed without notice.

Returns the pad name as a mortal SV.

```
SV *  PadnameSV(PADNAME * pn)
```

"PadnameUTF8"

NOTE: "PadnameUTF8" is experimental and may change or be removed without notice.

Whether PadnamePV is in UTF-8. Currently, this is always true.

```
bool  PadnameUTF8(PADNAME * pn)
```

"pad_new"

Create a new padlist, updating the global variables for the currently-compiling padlist to point to the new padlist. The following flags can be OR'ed together:

```
padnew_CLONE        this pad is for a cloned CV
padnew_SAVE         save old globals on the save stack
padnew_SAVESUB      also save extra stuff for start of sub
```

```
PADLIST*  pad_new(int flags)
```

"PL_comppad"

NOTE: "PL_comppad" is experimental and may change or be removed without notice.

During compilation, this points to the array containing the values part of the pad for the currently-compiling code. (At runtime a CV may have many such value arrays; at compile time just one is constructed.) At runtime, this points to the array containing the currently-relevant values for the pad for the currently-executing code.

"PL_comppad_name"

NOTE: "PL_comppad_name" is experimental and may change or be removed without notice.

During compilation, this points to the array containing the names part of the pad for the currently-compiling code.

"PL_curpad"

NOTE: "PL_curpad" is experimental and may change or be removed without notice.

Points directly to the body of the "PL_comppad" array. (I.e., this is "PadARRAY(PL_comppad)".)

Password and Group access

"GRPASSWD"

This symbol, if defined, indicates to the C program that "struct group" in grp.h contains "gr_passwd".

"HAS_ENDGRENT"

This symbol, if defined, indicates that the getgrent routine is available for finalizing sequential access of the group database.

"HAS_ENDGRENT_R"

This symbol, if defined, indicates that the "endgrent_r" routine is available to endgrent re-entrantly.

"HAS_ENDPWENT"

This symbol, if defined, indicates that the getgrent routine is available for finalizing sequential access of the passwd database.

"HAS_ENDPWENT_R"

This symbol, if defined, indicates that the "endpwent_r" routine is available to endpwent re-entrantly.

"HAS_GETGRENT"

This symbol, if defined, indicates that the "getgrent" routine is available for sequential access of the group database.

"HAS_GETGRENT_R"

This symbol, if defined, indicates that the "getgrent_r" routine is available to getgrent re-entrantly.

"HAS_GETPWENT"

This symbol, if defined, indicates that the "getpwent" routine is available for sequential access of the passwd database. If this is not available, the older "getpw()" function may be available.

"HAS_GETPWENT_R"

This symbol, if defined, indicates that the "getpwent_r" routine is available to getpwent re-entrantly.

"HAS_SETGRENT"

This symbol, if defined, indicates that the "setgrent" routine is available for initializing sequential access of the group database.

"HAS_SETGRENT_R"

This symbol, if defined, indicates that the "setgrent_r" routine is available to setgrent re-entrantly.

"HAS_SETPWENT"

This symbol, if defined, indicates that the "setpwent" routine is available for initializing sequential access of the passwd database.

"HAS_SETPWENT_R"

This symbol, if defined, indicates that the "setpwent_r" routine is available to setpwent re-entrantly.

"PWAGE"

This symbol, if defined, indicates to the C program that "struct passwd" contains "pw_age".

"PWCHANGE"

This symbol, if defined, indicates to the C program that "struct passwd" contains "pw_change".

"PWCLASS"

This symbol, if defined, indicates to the C program that "struct passwd" contains "pw_class".

"PWCOMMENT"

This symbol, if defined, indicates to the C program that "struct passwd" contains "pw_comment".

"PWEXPIRE"

This symbol, if defined, indicates to the C program that "struct passwd" contains "pw_expire".

"PWGECOS"

This symbol, if defined, indicates to the C program that "struct passwd" contains "pw_gecos".

"PWPASSWD"

This symbol, if defined, indicates to the C program that "struct passwd" contains "pw_passwd".

"PWQUOTA"

This symbol, if defined, indicates to the C program that "struct passwd" contains "pw_quota".

Paths to system commands

"CSH"

This symbol, if defined, contains the full pathname of csh.

"LOC_SED"

This symbol holds the complete pathname to the sed program.

"SH_PATH"

This symbol contains the full pathname to the shell used on this on this system to execute Bourne shell scripts. Usually, this will be /bin/sh, though it's possible that some systems will have /bin/ksh, /bin/pdksh, /bin/ash, /bin/bash, or even something such as D:/bin/sh.exe.

Prototype information

"CRYPT_R_PROTO"

This symbol encodes the prototype of "crypt_r". It is zero if "d_crypt_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_crypt_r" is defined.

"CTERMID_R_PROTO"

This symbol encodes the prototype of "ctermid_r". It is zero if "d_ctermid_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_ctermid_r" is defined.

"DRAND48_R_PROTO"

This symbol encodes the prototype of "drand48_r". It is zero if "d_drand48_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_drand48_r" is defined.

"ENDGRENT_R_PROTO"

This symbol encodes the prototype of "endgrent_r". It is zero if "d_endgrent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_endgrent_r" is defined.

"ENDHOSTENT_R_PROTO"

This symbol encodes the prototype of "endhostent_r". It is zero if "d_endhostent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_endhostent_r" is defined.

"ENDNETENT_R_PROTO"

This symbol encodes the prototype of "endnetent_r". It is zero if "d_endnetent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_endnetent_r" is defined.

"ENDPROTOENT_R_PROTO"

This symbol encodes the prototype of "endprotoent_r". It is zero if "d_endprotoent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_endprotoent_r" is defined.

"ENDPWENT_R_PROTO"

This symbol encodes the prototype of "endpwent_r". It is zero if "d_endpwent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_endpwent_r" is defined.

"ENDSERVENT_R_PROTO"

This symbol encodes the prototype of "endservent_r". It is zero if "d_endservent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_endservent_r" is defined.

"GDBMNDBM_H_USES_PROTOTYPES"

This symbol, if defined, indicates that gdbm/ndbm.h uses real "ANSI" C prototypes instead of K&R style function declarations without any parameter information. While "ANSI" C prototypes are supported in C++, K&R style function declarations will yield errors.

"GDBM_NDBM_H_USES_PROTOTYPES"

This symbol, if defined, indicates that <gdbm-ndbm.h> uses real "ANSI" C prototypes instead of K&R style function declarations without any parameter information. While "ANSI" C prototypes are supported in C++, K&R style function declarations will yield errors.

"GETGRENT_R_PROTO"

This symbol encodes the prototype of "getgrent_r". It is zero if "d_getgrent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getgrent_r" is defined.

"GETGRGID_R_PROTO"

This symbol encodes the prototype of "getgrgid_r". It is zero if "d_getgrgid_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getgrgid_r" is defined.

"GETGRNAM_R_PROTO"

This symbol encodes the prototype of "getgrnam_r". It is zero if "d_getgrnam_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getgrnam_r" is defined.

"GETHOSTBYADDR_R_PROTO"

This symbol encodes the prototype of "gethostbyaddr_r". It is zero if "d_gethostbyaddr_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_gethostbyaddr_r" is defined.

"GETHOSTBYNAME_R_PROTO"

This symbol encodes the prototype of "gethostbyname_r". It is zero if "d_gethostbyname_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_gethostbyname_r" is defined.

"GETHOSTENT_R_PROTO"

This symbol encodes the prototype of "gethostent_r". It is zero if "d_gethostent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_gethostent_r" is defined.

"GETLOGIN_R_PROTO"

This symbol encodes the prototype of "getlogin_r". It is zero if "d_getlogin_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getlogin_r" is defined.

"GETNETBYADDR_R_PROTO"

This symbol encodes the prototype of "getnetbyaddr_r". It is zero if "d_getnetbyaddr_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getnetbyaddr_r" is defined.

"GETNETBYNAME_R_PROTO"

This symbol encodes the prototype of "getnetbyname_r". It is zero if "d_getnetbyname_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getnetbyname_r" is defined.

"GETNETENT_R_PROTO"

This symbol encodes the prototype of "getnetent_r". It is zero if "d_getnetent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getnetent_r" is defined.

"GETPROTOBYNAME_R_PROTO"

This symbol encodes the prototype of "getprotobyname_r". It is zero if "d_getprotobyname_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getprotobyname_r" is defined.

"GETPROTOBYNUMBER_R_PROTO"

This symbol encodes the prototype of "getprotobynumber_r". It is zero if "d_getprotobynumber_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getprotobynumber_r" is defined.

"GETPROTOENT_R_PROTO"

This symbol encodes the prototype of "getprotoent_r". It is zero if "d_getprotoent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getprotoent_r" is defined.

"GETPWENT_R_PROTO"

This symbol encodes the prototype of "getpwent_r". It is zero if "d_getpwent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getpwent_r" is defined.

"GETPWNAM_R_PROTO"

This symbol encodes the prototype of "getpwnam_r". It is zero if "d_getpwnam_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getpwnam_r" is defined.

"GETPWUID_R_PROTO"

This symbol encodes the prototype of "getpwuid_r". It is zero if "d_getpwuid_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getpwuid_r" is defined.

"GETSERVBYNAME_R_PROTO"

This symbol encodes the prototype of "getservbyname_r". It is zero if "d_getservbyname_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getservbyname_r" is defined.

"GETSERVBYPORT_R_PROTO"

This symbol encodes the prototype of "getservbyport_r". It is zero if "d_getservbyport_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getservbyport_r" is defined.

"GETSERVENT_R_PROTO"

This symbol encodes the prototype of "getservent_r". It is zero if "d_getservent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getservent_r" is defined.

"GETSPNAM_R_PROTO"

This symbol encodes the prototype of "getspnam_r". It is zero if "d_getspnam_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_getspnam_r" is defined.

"HAS_DBMINIT_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the

"dbminit()" function. Otherwise, it is up to the program to supply one. A good guess is

 extern int dbminit(char *);

"HAS_DRAND48_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "drand48()" function. Otherwise, it is up to the program to supply one. A good guess is

 extern double drand48(void);

"HAS_FLOCK_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "flock()" function. Otherwise, it is up to the program to supply one. A good guess is

 extern int flock(int, int);

"HAS_GETHOST_PROTOS"

This symbol, if defined, indicates that netdb.h includes prototypes for "gethostent()", "gethostbyname()", and "gethostbyaddr()". Otherwise, it is up to the program to guess them. See netdbtype.U (part of metaconfig) for probing for various "Netdb_xxx_t" types.

"HAS_GETNET_PROTOS"

This symbol, if defined, indicates that netdb.h includes prototypes for "getnetent()", "getnetbyname()", and "getnetbyaddr()". Otherwise, it is up to the program to guess them. See netdbtype.U (part of metaconfig) for probing for various "Netdb_xxx_t" types.

"HAS_GETPROTO_PROTOS"

This symbol, if defined, indicates that netdb.h includes prototypes for "getprotoent()", "getprotobyname()", and "getprotobyaddr()". Otherwise, it is up to the program to guess them. See netdbtype.U (part of metaconfig) for probing for various "Netdb_xxx_t" types.

"HAS_GETSERV_PROTOS"

This symbol, if defined, indicates that netdb.h includes prototypes for "getservent()", "getservbyname()", and "getservbyaddr()". Otherwise, it is up to the program to guess them. See netdbtype.U (part of metaconfig) for probing for various "Netdb_xxx_t" types.

"HAS_MODFL_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "modfl()" function. Otherwise, it is up to the program to supply one.

"HAS_SBRK_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "sbrk()" function. Otherwise, it is up to the program to supply one. Good guesses are

```
extern void* sbrk(int);
extern void* sbrk(size_t);
```

"HAS_SETRESGID_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "setresgid()" function. Otherwise, it is up to the program to supply one. Good guesses are

```
extern int setresgid(uid_t ruid, uid_t euid, uid_t suid);
```

"HAS_SETRESUID_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "setresuid()" function. Otherwise, it is up to the program to supply one. Good guesses are

```
extern int setresuid(uid_t ruid, uid_t euid, uid_t suid);
```

"HAS_SHMAT_PROTOTYPE"

This symbol, if defined, indicates that the sys/shm.h includes a prototype for "shmat()". Otherwise, it is up to the program to guess one. "Shmat_t" "shmat(int, Shmat_t, int)" is a good guess, but not always right so it should be emitted by the program only when "HAS_SHMAT_PROTOTYPE" is not defined to avoid conflicting defs.

"HAS_SOCKATMARK_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "sockatmark()" function. Otherwise, it is up to the program to supply one. A good guess is

```
extern int sockatmark(int);
```

"HAS_SYSCALL_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "syscall()" function. Otherwise, it is up to the program to supply one. Good guesses are

extern int syscall(int, ...);

extern int syscall(long, ...);

"HAS_TELLDIR_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "telldir()" function. Otherwise, it is up to the program to supply one. A good guess is

extern long telldir(DIR*);

"NDBM_H_USES_PROTOTYPES"

This symbol, if defined, indicates that ndbm.h uses real "ANSI" C prototypes instead of K&R style function declarations without any parameter information. While "ANSI" C prototypes are supported in C++, K&R style function declarations will yield errors.

"RANDOM_R_PROTO"

This symbol encodes the prototype of "random_r". It is zero if "d_random_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_random_r" is defined.

"READDIR_R_PROTO"

This symbol encodes the prototype of "readdir_r". It is zero if "d_readdir_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_readdir_r" is defined.

"SETGRENT_R_PROTO"

This symbol encodes the prototype of "setgrent_r". It is zero if "d_setgrent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_setgrent_r" is defined.

"SETHOSTENT_R_PROTO"

This symbol encodes the prototype of "sethostent_r". It is zero if "d_sethostent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_sethostent_r" is defined.

"SETLOCALE_R_PROTO"

This symbol encodes the prototype of "setlocale_r". It is zero if "d_setlocale_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_setlocale_r" is defined.

"SETNETENT_R_PROTO"

This symbol encodes the prototype of "setnetent_r". It is zero if "d_setnetent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_setnetent_r" is

defined.

"SETPROTOENT_R_PROTO"

This symbol encodes the prototype of "setprotoent_r". It is zero if "d_setprotoent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_setprotoent_r" is defined.

"SETPWENT_R_PROTO"

This symbol encodes the prototype of "setpwent_r". It is zero if "d_setpwent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_setpwent_r" is defined.

"SETSERVENT_R_PROTO"

This symbol encodes the prototype of "setservent_r". It is zero if "d_setservent_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_setservent_r" is defined.

"SRAND48_R_PROTO"

This symbol encodes the prototype of "srand48_r". It is zero if "d_srand48_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_srand48_r" is defined.

"SRANDOM_R_PROTO"

This symbol encodes the prototype of "srandom_r". It is zero if "d_srandom_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_srandom_r" is defined.

"STRERROR_R_PROTO"

This symbol encodes the prototype of "strerror_r". It is zero if "d_strerror_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_strerror_r" is defined.

"TMPNAM_R_PROTO"

This symbol encodes the prototype of "tmpnam_r". It is zero if "d_tmpnam_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_tmpnam_r" is defined.

"TTYNAME_R_PROTO"

This symbol encodes the prototype of "ttyname_r". It is zero if "d_ttyname_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_ttyname_r" is defined.

"pregcomp"

Described in perlreguts.

```
REGEXP*  pregcomp(SV * const pattern, const U32 flags)
```

"pregexec"

Described in perlreguts.

```
I32  pregexec(REGEXP * const prog, char* stringarg, char* strend,
              char* strbeg, SSize_t minend, SV* screamer,
              U32 nosave)
```

"re_dup_guts"

Duplicate a regexp.

This routine is expected to clone a given regexp structure. It is only compiled under USE_ITHREADS.

After all of the core data stored in struct regexp is duplicated the "regexp_engine.dupe" method is used to copy any private data stored in the *pprivate pointer. This allows extensions to handle any duplication they need to do.

```
void  re_dup_guts(const REGEXP *sstr, REGEXP *dstr,
                  CLONE_PARAMS* param)
```

"regmatch_info"

Some basic information about the current match that is created by Perl_regexec_flags and then passed to regtry(), regmatch() etc. It is allocated as a local var on the stack, so nothing should be stored in it that needs preserving or clearing up on croak(). For that, see the aux_info and aux_info_eval members of the regmatch_state union.

"SvRX"

Convenience macro to get the REGEXP from a SV. This is approximately equivalent to the following snippet:

```
if (SvMAGICAL(sv))
    mg_get(sv);
if (SvROK(sv))
    sv = MUTABLE_SV(SvRV(sv));
if (SvTYPE(sv) == SVt_REGEXP)
    return (REGEXP*) sv;
```

"NULL" will be returned if a REGEXP* is not found.

REGEXP * SvRX(SV *sv)

"SvRXOK"

Returns a boolean indicating whether the SV (or the one it references) is a REGEXP.

If you want to do something with the REGEXP* later use SvRX instead and check for NULL.

bool SvRXOK(SV* sv)

Signals

"HAS_SIGINFO_SI_ADDR"

This symbol, if defined, indicates that "siginfo_t" has the "si_addr" member

"HAS_SIGINFO_SI_BAND"

This symbol, if defined, indicates that "siginfo_t" has the "si_band" member

"HAS_SIGINFO_SI_ERRNO"

This symbol, if defined, indicates that "siginfo_t" has the "si_errno" member

"HAS_SIGINFO_SI_PID"

This symbol, if defined, indicates that "siginfo_t" has the "si_pid" member

"HAS_SIGINFO_SI_STATUS"

This symbol, if defined, indicates that "siginfo_t" has the "si_status" member

"HAS_SIGINFO_SI_UID"

This symbol, if defined, indicates that "siginfo_t" has the "si_uid" member

"HAS_SIGINFO_SI_VALUE"

This symbol, if defined, indicates that "siginfo_t" has the "si_value" member

"PERL_SIGNALS_UNSAFE_FLAG"

If this bit in "PL_signals" is set, the system is uing the pre-Perl 5.8 unsafe signals. See "PERL_SIGNALS" in perlrun and "Deferred Signals (Safe Signals)" in perlipc.

U32 PERL_SIGNALS_UNSAFE_FLAG

"rsignal"

A wrapper for the C library signal(2). Don't use the latter, as the Perl version knows things that interact with the rest of the perl interpreter.

Sighandler_t rsignal(int i, Sighandler_t t)

"Sigjmp_buf"

This is the buffer type to be used with Sigsetjmp and Siglongjmp.

"Siglongjmp"

This macro is used in the same way as "siglongjmp()", but will invoke traditional "longjmp()" if siglongjmp isn't available. See "HAS_SIGSETJMP".

```
void  Siglongjmp(jmp_buf env, int val)
```

"SIG_NAME"

This symbol contains a list of signal names in order of signal number. This is intended to be used as a static array initialization, like this:

```
char *sig_name[] = { SIG_NAME };
```

The signals in the list are separated with commas, and each signal is surrounded by double quotes. There is no leading "SIG" in the signal name, i.e. "SIGQUIT" is known as ""QUIT"". Gaps in the signal numbers (up to "NSIG") are filled in with "NUMnn", etc., where nn is the actual signal number (e.g. "NUM37"). The signal number for "sig_name[i]" is stored in "sig_num[i]". The last element is 0 to terminate the list with a "NULL". This corresponds to the 0 at the end of the "sig_name_init" list. Note that this variable is initialized from the "sig_name_init", not from "sig_name" (which is unused).

"SIG_NUM"

This symbol contains a list of signal numbers, in the same order as the "SIG_NAME" list. It is suitable for static array initialization, as in:

```
int sig_num[] = { SIG_NUM };
```

The signals in the list are separated with commas, and the indices within that list and the "SIG_NAME" list match, so it's easy to compute the signal name from a number or vice versa at the price of a small dynamic linear lookup. Duplicates are allowed, but are moved to the end of the list. The signal number corresponding to "sig_name[i]" is "sig_number[i]". if (i < "NSIG") then "sig_number[i]" == i. The last element is 0, corresponding to the 0 at the end of the "sig_name_init" list. Note that this variable is initialized from the "sig_num_init", not from "sig_num" (which is unused).

"Sigsetjmp"

This macro is used in the same way as "sigsetjmp()", but will invoke traditional "setjmp()" if sigsetjmp isn't available. See "HAS_SIGSETJMP".

```
int  Sigsetjmp(jmp_buf env, int savesigs)
```

"SIG_SIZE"

This variable contains the number of elements of the "SIG_NAME" and "SIG_NUM" arrays,

excluding the final "NULL" entry.

"whichsig"

"whichsig_pv"

"whichsig_pvn"

"whichsig_sv"

These all convert a signal name into its corresponding signal number; returning -1 if no corresponding number was found.

They differ only in the source of the signal name:

"whichsig_pv" takes the name from the "NUL"-terminated string starting at "sig".

"whichsig" is merely a different spelling, a synonym, of "whichsig_pv".

"whichsig_pvn" takes the name from the string starting at "sig", with length "len" bytes.

"whichsig_sv" takes the name from the PV stored in the SV "sigsv".

```
I32  whichsig    (const char* sig)
I32  whichsig_pv (const char* sig)
I32  whichsig_pvn(const char* sig, STRLEN len)
I32  whichsig_sv (SV* sigsv)
```

Site configuration

These variables give details as to where various libraries, installation destinations, etc., go, as well as what various installation options were selected

"ARCHLIB"

This variable, if defined, holds the name of the directory in which the user wants to put architecture-dependent public library files for perl5. It is most often a local directory such as /usr/local/lib. Programs using this variable must be prepared to deal with filename expansion. If "ARCHLIB" is the same as "PRIVLIB", it is not defined, since presumably the program already searches "PRIVLIB".

"ARCHLIB_EXP"

This symbol contains the ~name expanded version of "ARCHLIB", to be used in programs that are not prepared to deal with ~ expansion at run-time.

"ARCHNAME"

This symbol holds a string representing the architecture name. It may be used to construct an architecture-dependant pathname where library files may be held under a private library, for instance.

"BIN"

This symbol holds the path of the bin directory where the package will be installed.

Program must be prepared to deal with ~name substitution.

"BIN_EXP"

This symbol is the filename expanded version of the "BIN" symbol, for programs that do not want to deal with that at run-time.

"INSTALL_USR_BIN_PERL"

This symbol, if defined, indicates that Perl is to be installed also as /usr/bin/perl.

"MULTIARCH"

This symbol, if defined, signifies that the build process will produce some binary files that are going to be used in a cross-platform environment. This is the case for example with the NeXT "fat" binaries that contain executables for several "CPUs".

"PERL_INC_VERSION_LIST"

This variable specifies the list of subdirectories in over which perl.c:"incpush()" and lib/lib.pm will automatically search when adding directories to @"INC", in a format suitable for a C initialization string. See the "inc_version_list" entry in Porting/Glossary for more details.

"PERL_OTHERLIBDIRS"

This variable contains a colon-separated set of paths for the perl binary to search for additional library files or modules. These directories will be tacked to the end of @"INC". Perl will automatically search below each path for version- and architecture-specific directories. See "PERL_INC_VERSION_LIST" for more details.

"PERL_RELOCATABLE_INC"

This symbol, if defined, indicates that we'd like to relocate entries in @"INC" at run time based on the location of the perl binary.

"PERL_TARGETARCH"

This symbol, if defined, indicates the target architecture Perl has been cross-compiled to. Undefined if not a cross-compile.

"PERL_USE_DEVEL"

This symbol, if defined, indicates that Perl was configured with "-Dusedevel", to enable development features. This should not be done for production builds.

"PERL_VENDORARCH"

If defined, this symbol contains the name of a private library. The library is

private in the sense that it needn't be in anyone's execution path, but it should be accessible by the world. It may have a ~ on the front. The standard distribution will put nothing in this directory. Vendors who distribute perl may wish to place their own architecture-dependent modules and extensions in this directory with

```
MakeMaker Makefile.PL INSTALLDIRS=vendor
```

or equivalent. See "INSTALL" for details.

"PERL_VENDORARCH_EXP"

This symbol contains the ~name expanded version of "PERL_VENDORARCH", to be used in programs that are not prepared to deal with ~ expansion at run-time.

"PERL_VENDORLIB_EXP"

This symbol contains the ~name expanded version of "VENDORLIB", to be used in programs that are not prepared to deal with ~ expansion at run-time.

"PERL_VENDORLIB_STEM"

This define is "PERL_VENDORLIB_EXP" with any trailing version-specific component removed. The elements in "inc_version_list" ("inc_version_list".U (part of metaconfig)) can be tacked onto this variable to generate a list of directories to search.

"PRIVLIB"

This symbol contains the name of the private library for this package. The library is private in the sense that it needn't be in anyone's execution path, but it should be accessible by the world. The program should be prepared to do ~ expansion.

"PRIVLIB_EXP"

This symbol contains the ~name expanded version of "PRIVLIB", to be used in programs that are not prepared to deal with ~ expansion at run-time.

"SITEARCH"

This symbol contains the name of the private library for this package. The library is private in the sense that it needn't be in anyone's execution path, but it should be accessible by the world. The program should be prepared to do ~ expansion. The standard distribution will put nothing in this directory. After perl has been installed, users may install their own local architecture-dependent modules in this directory with

```
MakeMaker Makefile.PL
```

or equivalent. See "INSTALL" for details.

"SITEARCH_EXP"

This symbol contains the ~name expanded version of "SITEARCH", to be used in programs that are not prepared to deal with ~ expansion at run-time.

"SITELIB"

This symbol contains the name of the private library for this package. The library is private in the sense that it needn't be in anyone's execution path, but it should be accessible by the world. The program should be prepared to do ~ expansion. The standard distribution will put nothing in this directory. After perl has been installed, users may install their own local architecture-independent modules in this directory with

MakeMaker Makefile.PL

or equivalent. See "INSTALL" for details.

"SITELIB_EXP"

This symbol contains the ~name expanded version of "SITELIB", to be used in programs that are not prepared to deal with ~ expansion at run-time.

"SITELIB_STEM"

This define is "SITELIB_EXP" with any trailing version-specific component removed. The elements in "inc_version_list" ("inc_version_list".U (part of metaconfig)) can be tacked onto this variable to generate a list of directories to search.

"STARTPERL"

This variable contains the string to put in front of a perl script to make sure (one hopes) that it runs with perl and not some shell.

"USE_64_BIT_ALL"

This symbol, if defined, indicates that 64-bit integers should be used when available. If not defined, the native integers will be used (be they 32 or 64 bits). The maximal possible 64-bitness is employed: LP64 or "ILP64", meaning that you will be able to use more than 2 gigabytes of memory. This mode is even more binary incompatible than "USE_64_BIT_INT". You may not be able to run the resulting executable in a 32-bit "CPU" at all or you may need at least to reboot your OS to 64-bit mode.

"USE_64_BIT_INT"

This symbol, if defined, indicates that 64-bit integers should be used when available. If not defined, the native integers will be employed (be they 32 or 64 bits). The minimal possible 64-bitness is used, just enough to get 64-bit integers into Perl.

This may mean using for example "long longs", while your memory may still be limited to 2 gigabytes.

"USE_BSD_GETPGRP"

This symbol, if defined, indicates that getpgrp needs one arguments whereas "USG" one needs none.

"USE_BSD_SETPGRP"

This symbol, if defined, indicates that setpgrp needs two arguments whereas "USG" one needs none. See also "HAS_SETPGID" for a "POSIX" interface.

"USE_CPLUSPLUS"

This symbol, if defined, indicates that a C++ compiler was used to compiled Perl and will be used to compile extensions.

"USE_CROSS_COMPILE"

This symbol, if defined, indicates that Perl is being cross-compiled.

"USE_C_BACKTRACE"

This symbol, if defined, indicates that Perl should be built with support for backtrace.

"USE_DTRACE"

This symbol, if defined, indicates that Perl should be built with support for DTrace.

"USE_DYNAMIC_LOADING"

This symbol, if defined, indicates that dynamic loading of some sort is available.

"USE_FAST_STDIO"

This symbol, if defined, indicates that Perl should be built to use 'fast stdio'. Defaults to define in Perls 5.8 and earlier, to undef later.

"USE_ITHREADS"

This symbol, if defined, indicates that Perl should be built to use the interpreter-based threading implementation.

"USE_KERN_PROC_PATHNAME"

This symbol, if defined, indicates that we can use sysctl with "KERN_PROC_PATHNAME" to get a full path for the executable, and hence convert $^X to an absolute path.

"USE_LARGE_FILES"

This symbol, if defined, indicates that large file support should be used when available.

"USE_LONG_DOUBLE"

This symbol, if defined, indicates that long doubles should be used when available.

"USE_MORE_BITS"

This symbol, if defined, indicates that 64-bit interfaces and long doubles should be used when available.

"USE_NSGETEXECUTABLEPATH"

This symbol, if defined, indicates that we can use "_NSGetExecutablePath" and realpath to get a full path for the executable, and hence convert $^X to an absolute path.

"USE_PERLIO"

This symbol, if defined, indicates that the PerlIO abstraction should be used throughout. If not defined, stdio should be used in a fully backward compatible manner.

"USE_QUADMATH"

This symbol, if defined, indicates that the quadmath library should be used when available.

"USE_REENTRANT_API"

This symbol, if defined, indicates that Perl should try to use the various "_r" versions of library functions. This is extremely experimental.

"USE_SEMCTL_SEMID_DS"

This symbol, if defined, indicates that "struct semid_ds" * is used for semctl "IPC_STAT".

"USE_SEMCTL_SEMUN"

This symbol, if defined, indicates that "union semun" is used for semctl "IPC_STAT".

"USE_SITECUSTOMIZE"

This symbol, if defined, indicates that sitecustomize should be used.

"USE_SOCKS"

This symbol, if defined, indicates that Perl should be built to use socks.

"USE_STAT_BLOCKS"

This symbol is defined if this system has a stat structure declaring "st_blksize" and "st_blocks".

"USE_STDIO_BASE"

This symbol is defined if the "_base" field (or similar) of the stdio "FILE" structure can be used to access the stdio buffer for a file handle. If this is defined, then the "FILE_base(fp)" macro will also be defined and should be used to access this

field. Also, the "FILE_bufsiz(fp)" macro will be defined and should be used to determine the number of bytes in the buffer. "USE_STDIO_BASE" will never be defined unless "USE_STDIO_PTR" is.

"USE_STDIO_PTR"

This symbol is defined if the "_ptr" and "_cnt" fields (or similar) of the stdio "FILE" structure can be used to access the stdio buffer for a file handle. If this is defined, then the "FILE_ptr(fp)" and "FILE_cnt(fp)" macros will also be defined and should be used to access these fields.

"USE_STRICT_BY_DEFAULT"

This symbol, if defined, enables additional defaults. At this time it only enables implicit strict by default.

"USE_THREADS"

This symbol, if defined, indicates that Perl should be built to use threads. At present, it is a synonym for and "USE_ITHREADS", but eventually the source ought to be changed to use this to mean "_any_" threading implementation.

Sockets configuration values

"HAS_SOCKADDR_IN6"

This symbol, if defined, indicates the availability of "struct sockaddr_in6";

"HAS_SOCKADDR_SA_LEN"

This symbol, if defined, indicates that the "struct sockaddr" structure has a member called "sa_len", indicating the length of the structure.

"HAS_SOCKADDR_STORAGE"

This symbol, if defined, indicates the availability of "struct sockaddr_storage";

"HAS_SOCKATMARK"

This symbol, if defined, indicates that the "sockatmark" routine is available to test whether a socket is at the out-of-band mark.

"HAS_SOCKET"

This symbol, if defined, indicates that the "BSD" "socket" interface is supported.

"HAS_SOCKETPAIR"

This symbol, if defined, indicates that the "BSD" "socketpair()" call is supported.

"HAS_SOCKS5_INIT"

This symbol, if defined, indicates that the "socks5_init" routine is available to initialize "SOCKS" 5.

"I_SOCKS"

This symbol, if defined, indicates that socks.h exists and should be included.

```
#ifdef I_SOCKS
    #include <socks.h>
#endif
```

"I_SYS_SOCKIO"

This symbol, if defined, indicates the sys/sockio.h should be included to get socket ioctl options, like "SIOCATMARK".

```
#ifdef I_SYS_SOCKIO
    #include <sys_sockio.h>
#endif
```

Source Filters

"filter_add"

Described in perlfilter.

```
SV*  filter_add(filter_t funcp, SV* datasv)
```

"filter_read"

Described in perlfilter.

```
I32  filter_read(int idx, SV *buf_sv, int maxlen)
```

Stack Manipulation Macros

"BHK"

Described in perlguts.

"BINOP"

Described in perlguts.

"DESTRUCTORFUNC_NOCONTEXT_t"

Described in perlguts.

"DESTRUCTORFUNC_t"

Described in perlguts.

"dMARK"

Declare a stack marker variable, "mark", for the XSUB. See "MARK" and "dORIGMARK".

```
dMARK;
```

"dORIGMARK"

Saves the original stack mark for the XSUB. See "ORIGMARK".

```
dORIGMARK;
```

"dSP"

Declares a local copy of perl's stack pointer for the XSUB, available via the "SP" macro. See "SP".

```
dSP;
```

"dTARGET"

Declare that this function uses "TARG"

```
dTARGET;
```

"EXTEND"

Used to extend the argument stack for an XSUB's return values. Once used, guarantees that there is room for at least "nitems" to be pushed onto the stack.

```
void  EXTEND(SP, SSize_t nitems)
```

"LISTOP"

Described in perlguts.

"LOGOP"

Described in perlguts.

"LOOP"

Described in perlguts.

"MARK"

Stack marker variable for the XSUB. See "dMARK".

"mPUSHi"

Push an integer onto the stack. The stack must have room for this element. Does not use "TARG". See also "PUSHi", "mXPUSHi" and "XPUSHi".

```
void  mPUSHi(IV iv)
```

"mPUSHn"

Push a double onto the stack. The stack must have room for this element. Does not use "TARG". See also "PUSHn", "mXPUSHn" and "XPUSHn".

```
void  mPUSHn(NV nv)
```

"mPUSHp"

Push a string onto the stack. The stack must have room for this element. The "len" indicates the length of the string. Does not use "TARG". See also "PUSHp", "mXPUSHp" and "XPUSHp".

```
void  mPUSHp(char* str, STRLEN len)
```

"mPUSHs"

Push an SV onto the stack and mortalizes the SV. The stack must have room for this element. Does not use "TARG". See also "PUSHs" and "mXPUSHs".

```
void  mPUSHs(SV* sv)
```

"mPUSHu"

Push an unsigned integer onto the stack. The stack must have room for this element. Does not use "TARG". See also "PUSHu", "mXPUSHu" and "XPUSHu".

```
void  mPUSHu(UV uv)
```

"mXPUSHi"

Push an integer onto the stack, extending the stack if necessary. Does not use "TARG". See also "XPUSHi", "mPUSHi" and "PUSHi".

```
void  mXPUSHi(IV iv)
```

"mXPUSHn"

Push a double onto the stack, extending the stack if necessary. Does not use "TARG". See also "XPUSHn", "mPUSHn" and "PUSHn".

```
void  mXPUSHn(NV nv)
```

"mXPUSHp"

Push a string onto the stack, extending the stack if necessary. The "len" indicates the length of the string. Does not use "TARG". See also "XPUSHp", "mPUSHp" and "PUSHp".

```
void  mXPUSHp(char* str, STRLEN len)
```

"mXPUSHs"

Push an SV onto the stack, extending the stack if necessary and mortalizes the SV. Does not use "TARG". See also "XPUSHs" and "mPUSHs".

```
void  mXPUSHs(SV* sv)
```

"mXPUSHu"

Push an unsigned integer onto the stack, extending the stack if necessary. Does not use "TARG". See also "XPUSHu", "mPUSHu" and "PUSHu".

```
void  mXPUSHu(UV uv)
```

"newXSproto"

Used by "xsubpp" to hook up XSUBs as Perl subs. Adds Perl prototypes to the subs.

"OP"

Described in perlguts.

"ORIGMARK"

The original stack mark for the XSUB. See "dORIGMARK".

"peep_t"

Described in perlguts.

"PL_runops"

Described in perlguts.

"PMOP"

Described in perlguts.

"POPi"

Pops an integer off the stack.

```
IV  POPi
```

"POPl"

Pops a long off the stack.

```
long  POPl
```

"POPn"

Pops a double off the stack.

```
NV  POPn
```

"POPp"

Pops a string off the stack.

```
char*  POPp
```

"POPpbytex"

Pops a string off the stack which must consist of bytes i.e. characters < 256.

```
char*  POPpbytex
```

"POPpx"

Pops a string off the stack. Identical to POPp. There are two names for historical reasons.

```
char*  POPpx
```

"POPs"

Pops an SV off the stack.

```
SV*  POPs
```

"POPu"

Pops an unsigned integer off the stack.

```
UV  POPu
```

"POPul"

Pops an unsigned long off the stack.

long POPul

"PUSHi"

Push an integer onto the stack. The stack must have room for this element. Handles 'set' magic. Uses "TARG", so "dTARGET" or "dXSTARG" should be called to declare it. Do not call multiple "TARG"-oriented macros to return lists from XSUB's - see "mPUSHi" instead. See also "XPUSHi" and "mXPUSHi".

void PUSHi(IV iv)

"PUSHMARK"

Opening bracket for arguments on a callback. See "PUTBACK" and perlcall.

void PUSHMARK(SP)

"PUSHmortal"

Push a new mortal SV onto the stack. The stack must have room for this element. Does not use "TARG". See also "PUSHs", "XPUSHmortal" and "XPUSHs".

void PUSHmortal

"PUSHn"

Push a double onto the stack. The stack must have room for this element. Handles 'set' magic. Uses "TARG", so "dTARGET" or "dXSTARG" should be called to declare it. Do not call multiple "TARG"-oriented macros to return lists from XSUB's - see "mPUSHn" instead. See also "XPUSHn" and "mXPUSHn".

void PUSHn(NV nv)

"PUSHp"

Push a string onto the stack. The stack must have room for this element. The "len" indicates the length of the string. Handles 'set' magic. Uses "TARG", so "dTARGET" or "dXSTARG" should be called to declare it. Do not call multiple "TARG"-oriented macros to return lists from XSUB's - see "mPUSHp" instead. See also "XPUSHp" and "mXPUSHp".

void PUSHp(char* str, STRLEN len)

"PUSHs"

Push an SV onto the stack. The stack must have room for this element. Does not handle 'set' magic. Does not use "TARG". See also "PUSHmortal", "XPUSHs", and "XPUSHmortal".

void PUSHs(SV* sv)

"PUSHu"

Push an unsigned integer onto the stack. The stack must have room for this element. Handles 'set' magic. Uses "TARG", so "dTARGET" or "dXSTARG" should be called to declare it. Do not call multiple "TARG"-oriented macros to return lists from XSUB's - see "mPUSHu" instead. See also "XPUSHu" and "mXPUSHu".

```
void  PUSHu(UV uv)
```

"PUTBACK"

Closing bracket for XSUB arguments. This is usually handled by "xsubpp". See "PUSHMARK" and perlcall for other uses.

```
PUTBACK;
```

"save_aptr"

Described in perlguts.

```
void  save_aptr(AV** aptr)
```

"save_ary"

Described in perlguts.

```
AV*  save_ary(GV* gv)
```

"SAVEBOOL"

Described in perlguts.

```
SAVEBOOL(bool i)
```

"SAVEDELETE"

Described in perlguts.

```
SAVEDELETE(HV * hv, char * key, I32 length)
```

"SAVEDESTRUCTOR"

Described in perlguts.

```
SAVEDESTRUCTOR(DESTRUCTORFUNC_NOCONTEXT_t f, void *p)
```

"SAVEDESTRUCTOR_X"

Described in perlguts.

```
SAVEDESTRUCTOR_X(DESTRUCTORFUNC_t f, void *p)
```

"SAVEFREEOP"

Described in perlguts.

```
SAVEFREEOP(OP *op)
```

"SAVEFREEPV"

Described in perlguts.

SAVEFREEPV(void * p)

"SAVEFREESV"

Described in perlguts.

SAVEFREESV(SV* sv)

"save_hash"

Described in perlguts.

HV* save_hash(GV* gv)

"save_hptr"

Described in perlguts.

void save_hptr(HV** hptr)

"SAVEI8"

Described in perlguts.

SAVEI8(I8 i)

"SAVEI32"

Described in perlguts.

SAVEI32(I32 i)

"SAVEI16"

Described in perlguts.

SAVEI16(I16 i)

"SAVEINT"

Described in perlguts.

SAVEINT(int i)

"save_item"

Described in perlguts.

void save_item(SV* item)

"SAVEIV"

Described in perlguts.

SAVEIV(IV i)

"save_list"

"DEPRECATED!" It is planned to remove "save_list" from a future release of Perl. Do not use it for new code; remove it from existing code.

Described in perlguts.

void save_list(SV** sarg, I32 maxsarg)

"SAVELONG"

Described in perlguts.

SAVELONG(long i)

"SAVEMORTALIZESV"

Described in perlguts.

SAVEMORTALIZESV(SV* sv)

"SAVEPPTR"

Described in perlguts.

SAVEPPTR(char * p)

"save_scalar"

Described in perlguts.

SV* save_scalar(GV* gv)

"SAVESPTR"

Described in perlguts.

SAVESPTR(SV * s)

"SAVESTACK_POS"

Described in perlguts.

SAVESTACK_POS()

"save_svref"

Described in perlguts.

SV* save_svref(SV** sptr)

"SP"

Stack pointer. This is usually handled by "xsubpp". See "dSP" and "SPAGAIN".

"SPAGAIN"

Refetch the stack pointer. Used after a callback. See perlcall.

SPAGAIN;

"TARG"

"TARG" is short for "target". It is an entry in the pad that an OPs "op_targ" refers to. It is scratchpad space, often used as a return value for the OP, but some use it for other purposes.

TARG;

"UNOP"

Described in perlguts.

"XPUSHi"

Push an integer onto the stack, extending the stack if necessary. Handles 'set' magic. Uses "TARG", so "dTARGET" or "dXSTARG" should be called to declare it. Do not call multiple "TARG"-oriented macros to return lists from XSUB's - see "mXPUSHi" instead. See also "PUSHi" and "mPUSHi".

```
void  XPUSHi(IV iv)
```

"XPUSHmortal"

Push a new mortal SV onto the stack, extending the stack if necessary. Does not use "TARG". See also "XPUSHs", "PUSHmortal" and "PUSHs".

```
void  XPUSHmortal
```

"XPUSHn"

Push a double onto the stack, extending the stack if necessary. Handles 'set' magic. Uses "TARG", so "dTARGET" or "dXSTARG" should be called to declare it. Do not call multiple "TARG"-oriented macros to return lists from XSUB's - see "mXPUSHn" instead. See also "PUSHn" and "mPUSHn".

```
void  XPUSHn(NV nv)
```

"XPUSHp"

Push a string onto the stack, extending the stack if necessary. The "len" indicates the length of the string. Handles 'set' magic. Uses "TARG", so "dTARGET" or "dXSTARG" should be called to declare it. Do not call multiple "TARG"-oriented macros to return lists from XSUB's - see "mXPUSHp" instead. See also "PUSHp" and "mPUSHp".

```
void  XPUSHp(char* str, STRLEN len)
```

"XPUSHs"

Push an SV onto the stack, extending the stack if necessary. Does not handle 'set' magic. Does not use "TARG". See also "XPUSHmortal", "PUSHs" and "PUSHmortal".

```
void  XPUSHs(SV* sv)
```

"XPUSHu"

Push an unsigned integer onto the stack, extending the stack if necessary. Handles 'set' magic. Uses "TARG", so "dTARGET" or "dXSTARG" should be called to declare it. Do not call multiple "TARG"-oriented macros to return lists from XSUB's - see "mXPUSHu" instead. See also "PUSHu" and "mPUSHu".

```
void  XPUSHu(UV uv)
```

"XS_APIVERSION_BOOTCHECK"

Macro to verify that the perl api version an XS module has been compiled against

matches the api version of the perl interpreter it's being loaded into.

```
  XS_APIVERSION_BOOTCHECK;
```

"XSRETURN"

Return from XSUB, indicating number of items on the stack. This is usually handled by

"xsubpp".

```
 void  XSRETURN(int nitems)
```

"XSRETURN_EMPTY"

Return an empty list from an XSUB immediately.

```
  XSRETURN_EMPTY;
```

"XSRETURN_IV"

Return an integer from an XSUB immediately. Uses "XST_mIV".

```
 void  XSRETURN_IV(IV iv)
```

"XSRETURN_NO"

Return &PL_sv_no from an XSUB immediately. Uses "XST_mNO".

```
  XSRETURN_NO;
```

"XSRETURN_NV"

Return a double from an XSUB immediately. Uses "XST_mNV".

```
 void  XSRETURN_NV(NV nv)
```

"XSRETURN_PV"

Return a copy of a string from an XSUB immediately. Uses "XST_mPV".

```
 void  XSRETURN_PV(char* str)
```

"XSRETURN_UNDEF"

Return &PL_sv_undef from an XSUB immediately. Uses "XST_mUNDEF".

```
  XSRETURN_UNDEF;
```

"XSRETURN_UV"

Return an integer from an XSUB immediately. Uses "XST_mUV".

```
 void  XSRETURN_UV(IV uv)
```

"XSRETURN_YES"

Return &PL_sv_yes from an XSUB immediately. Uses "XST_mYES".

```
  XSRETURN_YES;
```

"XST_mIV"

Place an integer into the specified position "pos" on the stack. The value is stored

in a new mortal SV.

void XST_mIV(int pos, IV iv)

"XST_mNO"

Place &PL_sv_no into the specified position "pos" on the stack.

void XST_mNO(int pos)

"XST_mNV"

Place a double into the specified position "pos" on the stack. The value is stored in a new mortal SV.

void XST_mNV(int pos, NV nv)

"XST_mPV"

Place a copy of a string into the specified position "pos" on the stack. The value is stored in a new mortal SV.

void XST_mPV(int pos, char* str)

"XST_mUNDEF"

Place &PL_sv_undef into the specified position "pos" on the stack.

void XST_mUNDEF(int pos)

"XST_mUV"

Place an unsigned integer into the specified position "pos" on the stack. The value is stored in a new mortal SV.

void XST_mUV(int pos, UV uv)

"XST_mYES"

Place &PL_sv_yes into the specified position "pos" on the stack.

void XST_mYES(int pos)

"XS_VERSION"

The version identifier for an XS module. This is usually handled automatically by "ExtUtils::MakeMaker". See "XS_VERSION_BOOTCHECK".

"XS_VERSION_BOOTCHECK"

Macro to verify that a PM module's $VERSION variable matches the XS module's "XS_VERSION" variable. This is usually handled automatically by "xsubpp". See "The VERSIONCHECK: Keyword" in perlxs.

XS_VERSION_BOOTCHECK;

String Handling

See also "Unicode Support".

"CAT2"

This macro concatenates 2 tokens together.

token CAT2(token x, token y)

"Copy"

The XSUB-writer's interface to the C "memcpy" function. The "src" is the source, "dest" is the destination, "nitems" is the number of items, and "type" is the type. May fail on overlapping copies. See also "Move".

void Copy(void* src, void* dest, int nitems, type)

"CopyD"

Like "Copy" but returns "dest". Useful for encouraging compilers to tail-call optimise.

void * CopyD(void* src, void* dest, int nitems, type)

"delimcpy"

Copy a source buffer to a destination buffer, stopping at (but not including) the first occurrence in the source of an unescaped (defined below) delimiter byte, "delim". The source is the bytes between "from"?and?"from_end"?-?1. Similarly, the dest is "to" up to "to_end".

The number of bytes copied is written to *retlen.

Returns the position of the first uncopied "delim" in the "from" buffer, but if there is no such occurrence before "from_end", then "from_end" is returned, and the entire buffer "from"?..?"from_end"?-?1 is copied.

If there is room in the destination available after the copy, an extra terminating safety "NUL" byte is appended (not included in the returned length).

The error case is if the destination buffer is not large enough to accommodate everything that should be copied. In this situation, a value larger than "to_end"?-?"to" is written to *retlen, and as much of the source as fits will be written to the destination. Not having room for the safety "NUL" is not considered an error.

In the following examples, let "x" be the delimiter, and 0 represent a "NUL" byte (NOT the digit 0). Then we would have

Source Destination

abcxdef abc0

provided the destination buffer is at least 4 bytes long.

An escaped delimiter is one which is immediately preceded by a single backslash.

Escaped delimiters are copied, and the copy continues past the delimiter; the backslash is not copied:

```
  Source       Destination
 abc\xdef      abcxdef0
```

(provided the destination buffer is at least 8 bytes long).

It's actually somewhat more complicated than that. A sequence of any odd number of backslashes escapes the following delimiter, and the copy continues with exactly one of the backslashes stripped.

```
     Source         Destination
      abc\xdef        abcxdef0
    abc\\\xdef       abc\\xdef0
   abc\\\\\xdef     abc\\\\xdef0
```

(as always, if the destination is large enough)

An even number of preceding backslashes does not escape the delimiter, so that the copy stops just before it, and includes all the backslashes (no stripping; zero is considered even):

```
      Source         Destination
       abcxdef         abc0
     abc\\xdef         abc\\0
    abc\\\\xdef        abc\\\\0
```

```
 char*  delimcpy(char* to, const char* to_end, const char* from,
                 const char* from_end, const int delim,
                 I32* retlen)
```

"fbm_compile"

Analyzes the string in order to make fast searches on it using "fbm_instr()" -- the Boyer-Moore algorithm.

```
 void  fbm_compile(SV* sv, U32 flags)
```

"fbm_instr"

Returns the location of the SV in the string delimited by "big" and "bigend" ("bigend") is the char following the last char). It returns "NULL" if the string can't be found. The "sv" does not have to be "fbm_compiled", but the search will not be as fast then.

```
 char*  fbm_instr(unsigned char* big, unsigned char* bigend,
                  SV* littlestr, U32 flags)
```

"foldEQ"

Returns true if the leading "len" bytes of the strings "s1" and "s2" are the same case-insensitively; false otherwise. Uppercase and lowercase ASCII range bytes match themselves and their opposite case counterparts. Non-cased and non-ASCII range bytes match only themselves.

```
 I32  foldEQ(const char* a, const char* b, I32 len)
```

"ibcmp"

This is a synonym for "(!?foldEQ())"

```
 I32  ibcmp(const char* a, const char* b, I32 len)
```

"ibcmp_locale"

This is a synonym for "(!?foldEQ_locale())"

```
 I32  ibcmp_locale(const char* a, const char* b, I32 len)
```

"ibcmp_utf8"

This is a synonym for "(!?foldEQ_utf8())"

```
 I32  ibcmp_utf8(const char *s1, char **pe1, UV l1, bool u1,
                 const char *s2, char **pe2, UV l2, bool u2)
```

"instr"

Same as strstr(3), which finds and returns a pointer to the first occurrence of the NUL-terminated substring "little" in the NUL-terminated string "big", returning NULL if not found. The terminating NUL bytes are not compared.

```
 char*  instr(const char* big, const char* little)
```

"memCHRs"

Returns the position of the first occurence of the byte "c" in the literal string "list", or NULL if "c" doesn't appear in "list". All bytes are treated as unsigned char. Thus this macro can be used to determine if "c" is in a set of particular characters. Unlike strchr(3), it works even if "c" is "NUL" (and the set doesn't include "NUL").

```
 bool  memCHRs("list", char c)
```

"memEQ"

Test two buffers (which may contain embedded "NUL" characters, to see if they are equal. The "len" parameter indicates the number of bytes to compare. Returns true or

false. It is undefined behavior if either of the buffers doesn't contain at least "len" bytes.

```
bool  memEQ(char* s1, char* s2, STRLEN len)
```

"memEQs"

Like "memEQ", but the second string is a literal enclosed in double quotes, "l1" gives the number of bytes in "s1". Returns true or false.

```
bool  memEQs(char* s1, STRLEN l1, "s2")
```

"memNE"

Test two buffers (which may contain embedded "NUL" characters, to see if they are not equal. The "len" parameter indicates the number of bytes to compare. Returns true or false. It is undefined behavior if either of the buffers doesn't contain at least "len" bytes.

```
bool  memNE(char* s1, char* s2, STRLEN len)
```

"memNEs"

Like "memNE", but the second string is a literal enclosed in double quotes, "l1" gives the number of bytes in "s1". Returns true or false.

```
bool  memNEs(char* s1, STRLEN l1, "s2")
```

"Move"

The XSUB-writer's interface to the C "memmove" function. The "src" is the source, "dest" is the destination, "nitems" is the number of items, and "type" is the type. Can do overlapping moves. See also "Copy".

```
void  Move(void* src, void* dest, int nitems, type)
```

"MoveD"

Like "Move" but returns "dest". Useful for encouraging compilers to tail-call optimise.

```
void *  MoveD(void* src, void* dest, int nitems, type)
```

"my_snprintf"

The C library "snprintf" functionality, if available and standards-compliant (uses "vsnprintf", actually). However, if the "vsnprintf" is not available, will unfortunately use the unsafe "vsprintf" which can overrun the buffer (there is an overrun check, but that may be too late). Consider using "sv_vcatpvf" instead, or getting "vsnprintf".

```
int  my_snprintf(char *buffer, const Size_t len,
```

const char *format, ...)

"my_sprintf"

"DEPRECATED!" It is planned to remove "my_sprintf" from a future release of Perl. Do not use it for new code; remove it from existing code.

Do NOT use this due to the possibility of overflowing "buffer". Instead use my_snprintf()

int my_sprintf(NN char *buffer, NN const char *pat, ...)

"my_strlcat"

The C library "strlcat" if available, or a Perl implementation of it. This operates on C "NUL"-terminated strings.

"my_strlcat()" appends string "src" to the end of "dst". It will append at most "size?-?strlen(dst)?-?1" characters. It will then "NUL"-terminate, unless "size" is 0 or the original "dst" string was longer than "size" (in practice this should not happen as it means that either "size" is incorrect or that "dst" is not a proper "NUL"-terminated string).

Note that "size" is the full size of the destination buffer and the result is guaranteed to be "NUL"-terminated if there is room. Note that room for the "NUL" should be included in "size".

The return value is the total length that "dst" would have if "size" is sufficiently large. Thus it is the initial length of "dst" plus the length of "src". If "size" is smaller than the return, the excess was not appended.

Size_t my_strlcat(char *dst, const char *src, Size_t size)

"my_strlcpy"

The C library "strlcpy" if available, or a Perl implementation of it. This operates on C "NUL"-terminated strings.

"my_strlcpy()" copies up to "size?-?1" characters from the string "src" to "dst", "NUL"-terminating the result if "size" is not 0.

The return value is the total length "src" would be if the copy completely succeeded. If it is larger than "size", the excess was not copied.

Size_t my_strlcpy(char *dst, const char *src, Size_t size)

"my_strnlen"

The C library "strnlen" if available, or a Perl implementation of it.

"my_strnlen()" computes the length of the string, up to "maxlen" characters. It will

never attempt to address more than "maxlen" characters, making it suitable for use with strings that are not guaranteed to be NUL-terminated.

```
Size_t  my_strnlen(const char *str, Size_t maxlen)
```

"my_vsnprintf"

The C library "vsnprintf" if available and standards-compliant. However, if the "vsnprintf" is not available, will unfortunately use the unsafe "vsprintf" which can overrun the buffer (there is an overrun check, but that may be too late). Consider using "sv_vcatpvf" instead, or getting "vsnprintf".

```
int  my_vsnprintf(char *buffer, const Size_t len,
                  const char *format, va_list ap)
```

"ninstr"

Find the first (leftmost) occurrence of a sequence of bytes within another sequence. This is the Perl version of "strstr()", extended to handle arbitrary sequences, potentially containing embedded "NUL" characters ("NUL" is what the initial "n" in the function name stands for; some systems have an equivalent, "memmem()", but with a somewhat different API).

Another way of thinking about this function is finding a needle in a haystack. "big" points to the first byte in the haystack. "big_end" points to one byte beyond the final byte in the haystack. "little" points to the first byte in the needle. "little_end" points to one byte beyond the final byte in the needle. All the parameters must be non-"NULL".

The function returns "NULL" if there is no occurrence of "little" within "big". If "little" is the empty string, "big" is returned.

Because this function operates at the byte level, and because of the inherent characteristics of UTF-8 (or UTF-EBCDIC), it will work properly if both the needle and the haystack are strings with the same UTF-8ness, but not if the UTF-8ness differs.

```
char*  ninstr(const char* big, const char* bigend,
              const char* little, const char* lend)
```

"Nullch"

Null character pointer. (No longer available when "PERL_CORE" is defined.)

"rninstr"

Like "ninstr", but instead finds the final (rightmost) occurrence of a sequence of bytes within another sequence, returning "NULL" if there is no such occurrence.

```
 char*  rninstr(const char* big, const char* bigend,
                const char* little, const char* lend)
```

"savepv"

Perl's version of "strdup()". Returns a pointer to a newly allocated string which is a duplicate of "pv". The size of the string is determined by "strlen()", which means it may not contain embedded "NUL" characters and must have a trailing "NUL". To prevent memory leaks, the memory allocated for the new string needs to be freed when no longer needed. This can be done with the "Safefree" function, or "SAVEFREEPV".

On some platforms, Windows for example, all allocated memory owned by a thread is deallocated when that thread ends. So if you need that not to happen, you need to use the shared memory functions, such as "savesharedpv".

```
 char*  savepv(const char* pv)
```

"savepvn"

Perl's version of what "strndup()" would be if it existed. Returns a pointer to a newly allocated string which is a duplicate of the first "len" bytes from "pv", plus a trailing "NUL" byte. The memory allocated for the new string can be freed with the "Safefree()" function.

On some platforms, Windows for example, all allocated memory owned by a thread is deallocated when that thread ends. So if you need that not to happen, you need to use the shared memory functions, such as "savesharedpvn".

```
 char*  savepvn(const char* pv, Size_t len)
```

"savepvs"

Like "savepvn", but takes a literal string instead of a string/length pair.

```
 char*  savepvs("literal string")
```

"savesharedpv"

A version of "savepv()" which allocates the duplicate string in memory which is shared between threads.

```
 char*  savesharedpv(const char* pv)
```

"savesharedpvn"

A version of "savepvn()" which allocates the duplicate string in memory which is shared between threads. (With the specific difference that a "NULL" pointer is not acceptable)

```
 char*  savesharedpvn(const char *const pv, const STRLEN len)
```

"savesharedpvs"

A version of "savepvs()" which allocates the duplicate string in memory which is shared between threads.

```
char*  savesharedpvs("literal string")
```

"savesharedsvpv"

A version of "savesharedpv()" which allocates the duplicate string in memory which is shared between threads.

```
char*  savesharedsvpv(SV *sv)
```

"savesvpv"

A version of "savepv()"/"savepvn()" which gets the string to duplicate from the passed in SV using "SvPV()"

On some platforms, Windows for example, all allocated memory owned by a thread is deallocated when that thread ends. So if you need that not to happen, you need to use the shared memory functions, such as "savesharedsvpv".

```
char*  savesvpv(SV* sv)
```

"strEQ"

Test two "NUL"-terminated strings to see if they are equal. Returns true or false.

```
bool  strEQ(char* s1, char* s2)
```

"strGE"

Test two "NUL"-terminated strings to see if the first, "s1", is greater than or equal to the second, "s2". Returns true or false.

```
bool  strGE(char* s1, char* s2)
```

"strGT"

Test two "NUL"-terminated strings to see if the first, "s1", is greater than the second, "s2". Returns true or false.

```
bool  strGT(char* s1, char* s2)
```

"STRINGIFY"

This macro surrounds its token with double quotes.

```
string  STRINGIFY(token x)
```

"strLE"

Test two "NUL"-terminated strings to see if the first, "s1", is less than or equal to the second, "s2". Returns true or false.

```
bool  strLE(char* s1, char* s2)
```

"strLT"

Test two "NUL"-terminated strings to see if the first, "s1", is less than the second, "s2". Returns true or false.

```
bool  strLT(char* s1, char* s2)
```

"strNE"

Test two "NUL"-terminated strings to see if they are different. Returns true or false.

```
bool  strNE(char* s1, char* s2)
```

"strnEQ"

Test two "NUL"-terminated strings to see if they are equal. The "len" parameter indicates the number of bytes to compare. Returns true or false. (A wrapper for "strncmp").

```
bool  strnEQ(char* s1, char* s2, STRLEN len)
```

"strnNE"

Test two "NUL"-terminated strings to see if they are different. The "len" parameter indicates the number of bytes to compare. Returns true or false. (A wrapper for "strncmp").

```
bool  strnNE(char* s1, char* s2, STRLEN len)
```

"STR_WITH_LEN"

Returns two comma separated tokens of the input literal string, and its length. This is convenience macro which helps out in some API calls. Note that it can't be used as an argument to macros or functions that under some configurations might be macros, which means that it requires the full Perl_xxx(aTHX_ ...) form for any API calls where it's used.

```
pair  STR_WITH_LEN("literal string")
```

"Zero"

The XSUB-writer's interface to the C "memzero" function. The "dest" is the destination, "nitems" is the number of items, and "type" is the type.

```
void  Zero(void* dest, int nitems, type)
```

"ZeroD"

Like "Zero" but returns dest. Useful for encouraging compilers to tail-call optimise.

```
void *  ZeroD(void* dest, int nitems, type)
```

"SVt_IV"

Type flag for scalars. See "svtype".

"SVt_NULL"

Type flag for scalars. See "svtype".

"SVt_NV"

Type flag for scalars. See "svtype".

"SVt_PV"

Type flag for scalars. See "svtype".

"SVt_PVAV"

Type flag for arrays. See "svtype".

"SVt_PVCV"

Type flag for subroutines. See "svtype".

"SVt_PVFM"

Type flag for formats. See "svtype".

"SVt_PVGV"

Type flag for typeglobs. See "svtype".

"SVt_PVHV"

Type flag for hashes. See "svtype".

"SVt_PVIO"

Type flag for I/O objects. See "svtype".

"SVt_PVIV"

Type flag for scalars. See "svtype".

"SVt_PVLV"

Type flag for scalars. See "svtype".

"SVt_PVMG"

Type flag for scalars. See "svtype".

"SVt_PVNV"

Type flag for scalars. See "svtype".

"SVt_REGEXP"

Type flag for regular expressions. See "svtype".

"svtype"

An enum of flags for Perl types. These are found in the file sv.h in the "svtype" enum. Test these flags with the "SvTYPE" macro.

The types are:

SVt_NULL

SVt_IV

SVt_NV

SVt_RV

SVt_PV

SVt_PVIV

SVt_PVNV

SVt_PVMG

SVt_INVLIST

SVt_REGEXP

SVt_PVGV

SVt_PVLV

SVt_PVAV

SVt_PVHV

SVt_PVCV

SVt_PVFM

SVt_PVIO

These are most easily explained from the bottom up.

"SVt_PVIO" is for I/O objects, "SVt_PVFM" for formats, "SVt_PVCV" for subroutines, "SVt_PVHV" for hashes and "SVt_PVAV" for arrays.

All the others are scalar types, that is, things that can be bound to a "$" variable.

For these, the internal types are mostly orthogonal to types in the Perl language.

Hence, checking "SvTYPE(sv) < SVt_PVAV" is the best way to see whether something is a scalar.

"SVt_PVGV" represents a typeglob. If "!SvFAKE(sv)", then it is a real, incoercible typeglob. If "SvFAKE(sv)", then it is a scalar to which a typeglob has been assigned. Assigning to it again will stop it from being a typeglob. "SVt_PVLV" represents a scalar that delegates to another scalar behind the scenes. It is used, e.g., for the return value of "substr" and for tied hash and array elements. It can hold any scalar value, including a typeglob. "SVt_REGEXP" is for regular expressions. "SVt_INVLIST" is for Perl core internal use only.

"SVt_PVMG" represents a "normal" scalar (not a typeglob, regular expression, or

delegate). Since most scalars do not need all the internal fields of a PVMG, we save memory by allocating smaller structs when possible. All the other types are just simpler forms of "SVt_PVMG", with fewer internal fields. "SVt_NULL" can only hold undef. "SVt_IV" can hold undef, an integer, or a reference. ("SVt_RV" is an alias for "SVt_IV", which exists for backward compatibility.) "SVt_NV" can hold any of those or a double. "SVt_PV" can only hold "undef" or a string. "SVt_PVIV" is a superset of "SVt_PV" and "SVt_IV". "SVt_PVNV" is similar. "SVt_PVMG" can hold anything "SVt_PVNV" can hold, but it can, but does not have to, be blessed or magical.

SV Handling

An SV (or AV, HV, etc.) is allocated in two parts: the head (struct sv, av, hv...) contains type and reference count information, and for many types, a pointer to the body (struct xrv, xpv, xpviv...), which contains fields specific to each type. Some types store all they need in the head, so don't have a body.

In all but the most memory-paranoid configurations (ex: PURIFY), heads and bodies are allocated out of arenas, which by default are approximately 4K chunks of memory parcelled up into N heads or bodies. Sv-bodies are allocated by their sv-type, guaranteeing size consistency needed to allocate safely from arrays.

For SV-heads, the first slot in each arena is reserved, and holds a link to the next arena, some flags, and a note of the number of slots. Snaked through each arena chain is a linked list of free items; when this becomes empty, an extra arena is allocated and divided up into N items which are threaded into the free list.

SV-bodies are similar, but they use arena-sets by default, which separate the link and info from the arena itself, and reclaim the 1st slot in the arena. SV-bodies are further described later.

The following global variables are associated with arenas:

```
 PL_sv_arenaroot     pointer to list of SV arenas
 PL_sv_root          pointer to list of free SV structures
 PL_body_arenas      head of linked-list of body arenas
 PL_body_roots[]     array of pointers to list of free bodies of svtype
                     arrays are indexed by the svtype needed
```

A few special SV heads are not allocated from an arena, but are instead directly created in the interpreter structure, eg PL_sv_undef. The size of arenas can be changed from the default by setting PERL_ARENA_SIZE appropriately at compile time.

The SV arena serves the secondary purpose of allowing still-live SVs to be located and destroyed during final cleanup.

At the lowest level, the macros new_SV() and del_SV() grab and free an SV head. (If debugging with -DD, del_SV() calls the function S_del_sv() to return the SV to the free list with error checking.) new_SV() calls more_sv() / sv_add_arena() to add an extra arena if the free list is empty. SVs in the free list have their SvTYPE field set to all ones.

At the time of very final cleanup, sv_free_arenas() is called from perl_destruct() to physically free all the arenas allocated since the start of the interpreter.

The internal function visit() scans the SV arenas list, and calls a specified function for each SV it finds which is still live, i.e. which has an SvTYPE other than all 1's, and a non-zero SvREFCNT. visit() is used by the following functions (specified as [function that calls visit()] / [function called by visit() for each SV]):

sv_report_used() / do_report_used()
dump all remaining SVs (debugging aid)

sv_clean_objs() / do_clean_objs(),do_clean_named_objs(),
do_clean_named_io_objs(),do_curse()
Attempt to free all objects pointed to by RVs,
try to do the same for all objects indir-
ectly referenced by typeglobs too, and
then do a final sweep, cursing any
objects that remain. Called once from
perl_destruct(), prior to calling sv_clean_all()
below.

sv_clean_all() / do_clean_all()
SvREFCNT_dec(sv) each remaining SV, possibly
triggering an sv_free(). It also sets the
SVf_BREAK flag on the SV to indicate that the
refcnt has been artificially lowered, and thus
stopping sv_free() from giving spurious warnings
about SVs which unexpectedly have a refcnt
of zero. called repeatedly from perl_destruct()
until there are no SVs left.

Private API to rest of sv.c

new_SV(), del_SV(),

new_XPVNV(), del_XPVGV(),

etc

Public API:

sv_report_used(), sv_clean_objs(), sv_clean_all(), sv_free_arenas()

"boolSV"

Returns a true SV if "b" is a true value, or a false SV if "b" is 0.

See also "PL_sv_yes" and "PL_sv_no".

```
 SV *  boolSV(bool b)
```

"croak_xs_usage"

A specialised variant of "croak()" for emitting the usage message for xsubs

```
    croak_xs_usage(cv, "eee_yow");
```

works out the package name and subroutine name from "cv", and then calls "croak()".

Hence if "cv" is &ouch::awk, it would call "croak" as:

```
 Perl_croak(aTHX_ "Usage: %" SVf "::%" SVf "(%s)", "ouch" "awk",
                                                 "eee_yow");
 void  croak_xs_usage(const CV *const cv, const char *const params)
```

"DEFSV"

Returns the SV associated with $_

```
 SV *  DEFSV
```

"DEFSV_set"

Associate "sv" with $_

```
 void  DEFSV_set(SV * sv)
```

"get_sv"

Returns the SV of the specified Perl scalar. "flags" are passed to "gv_fetchpv". If "GV_ADD" is set and the Perl variable does not exist then it will be created. If "flags" is zero and the variable does not exist then NULL is returned.

NOTE: the "perl_get_sv()" form is deprecated.

```
 SV*  get_sv(const char *name, I32 flags)
```

"isGV_with_GP"

Returns a boolean as to whether or not "sv" is a GV with a pointer to a GP (glob pointer).

bool isGV_with_GP(SV * sv)

"looks_like_number"

Test if the content of an SV looks like a number (or is a number). "Inf" and "Infinity" are treated as numbers (so will not issue a non-numeric warning), even if your "atof()" doesn't grok them. Get-magic is ignored.

I32 looks_like_number(SV *const sv)

"MUTABLE_PTR"

"MUTABLE_AV"

"MUTABLE_CV"

"MUTABLE_GV"

"MUTABLE_HV"

"MUTABLE_IO"

"MUTABLE_SV"

The "MUTABLE_*"() macros cast pointers to the types shown, in such a way (compiler permitting) that casting away const-ness will give a warning; e.g.:

```
const SV *sv = ...;
AV *av1 = (AV*)sv;        <== BAD:  the const has been silently
                              cast away
AV *av2 = MUTABLE_AV(sv); <== GOOD: it may warn
```

"MUTABLE_PTR" is the base macro used to derive new casts. The other already-built-in ones return pointers to what their names indicate.

```
void *  MUTABLE_PTR(void * p)
AV *    MUTABLE_AV (AV * p)
CV *    MUTABLE_CV (CV * p)
GV *    MUTABLE_GV (GV * p)
HV *    MUTABLE_HV (HV * p)
IO *    MUTABLE_IO (IO * p)
SV *    MUTABLE_SV (SV * p)
```

"newRV"

"newRV_inc"

These are identical. They create an RV wrapper for an SV. The reference count for the original SV is incremented.

SV* newRV(SV *const sv)

"newRV_noinc"

Creates an RV wrapper for an SV. The reference count for the original SV is not incremented.

```
SV*  newRV_noinc(SV *const tmpRef)
```

"newSV"

Creates a new SV. A non-zero "len" parameter indicates the number of bytes of preallocated string space the SV should have. An extra byte for a trailing "NUL" is also reserved. ("SvPOK" is not set for the SV even if string space is allocated.) The reference count for the new SV is set to 1.

In 5.9.3, "newSV()" replaces the older "NEWSV()" API, and drops the first parameter, x, a debug aid which allowed callers to identify themselves. This aid has been superseded by a new build option, "PERL_MEM_LOG" (see "PERL_MEM_LOG" in perlhacktips). The older API is still there for use in XS modules supporting older perls.

```
SV*  newSV(const STRLEN len)
```

"newSVhek"

Creates a new SV from the hash key structure. It will generate scalars that point to the shared string table where possible. Returns a new (undefined) SV if "hek" is NULL.

```
SV*  newSVhek(const HEK *const hek)
```

"newSViv"

Creates a new SV and copies an integer into it. The reference count for the SV is set to 1.

```
SV*  newSViv(const IV i)
```

"newSVnv"

Creates a new SV and copies a floating point value into it. The reference count for the SV is set to 1.

```
SV*  newSVnv(const NV n)
```

"newSVpadname"

NOTE: "newSVpadname" is experimental and may change or be removed without notice.

Creates a new SV containing the pad name.

```
SV*  newSVpadname(PADNAME *pn)
```

"newSVpv"

Creates a new SV and copies a string (which may contain "NUL" ("\0") characters) into

it. The reference count for the SV is set to 1. If "len" is zero, Perl will compute the length using "strlen()", (which means if you use this option, that "s" can't have embedded "NUL" characters and has to have a terminating "NUL" byte).

This function can cause reliability issues if you are likely to pass in empty strings that are not null terminated, because it will run strlen on the string and potentially run past valid memory.

Using "newSVpvn" is a safer alternative for non "NUL" terminated strings. For string literals use "newSVpvs" instead. This function will work fine for "NUL" terminated strings, but if you want to avoid the if statement on whether to call "strlen" use "newSVpvn" instead (calling "strlen" yourself).

```
SV*  newSVpv(const char *const s, const STRLEN len)
```

"newSVpvf"

Creates a new SV and initializes it with the string formatted like "sv_catpvf".

NOTE: "newSVpvf" must be explicitly called as "Perl_newSVpvf" with an "aTHX_" parameter.

```
SV*  Perl_newSVpvf(pTHX_ const char *const pat, ...)
```

"newSVpvf_nocontext"

Like "newSVpvf" but does not take a thread context ("aTHX") parameter, so is used in situations where the caller doesn't already have the thread context.

```
SV*  newSVpvf_nocontext(const char *const pat, ...)
```

"newSVpvn"

Creates a new SV and copies a string into it, which may contain "NUL" characters ("\0") and other binary data. The reference count for the SV is set to 1. Note that if "len" is zero, Perl will create a zero length (Perl) string. You are responsible for ensuring that the source buffer is at least "len" bytes long. If the "buffer" argument is NULL the new SV will be undefined.

```
SV*  newSVpvn(const char *const buffer, const STRLEN len)
```

"newSVpvn_flags"

Creates a new SV and copies a string (which may contain "NUL" ("\0") characters) into it. The reference count for the SV is set to 1. Note that if "len" is zero, Perl will create a zero length string. You are responsible for ensuring that the source string is at least "len" bytes long. If the "s" argument is NULL the new SV will be undefined. Currently the only flag bits accepted are "SVf_UTF8" and "SVs_TEMP". If

"SVs_TEMP" is set, then "sv_2mortal()" is called on the result before returning. If "SVf_UTF8" is set, "s" is considered to be in UTF-8 and the "SVf_UTF8" flag will be set on the new SV. "newSVpvn_utf8()" is a convenience wrapper for this function, defined as

```
#define newSVpvn_utf8(s, len, u)                    \
    newSVpvn_flags((s), (len), (u) ? SVf_UTF8 : 0)
```

```
SV*  newSVpvn_flags(const char *const s, const STRLEN len,
                    const U32 flags)
```

"newSVpvn_share"

Creates a new SV with its "SvPVX_const" pointing to a shared string in the string table. If the string does not already exist in the table, it is created first. Turns on the "SvIsCOW" flag (or "READONLY" and "FAKE" in 5.16 and earlier). If the "hash" parameter is non-zero, that value is used; otherwise the hash is computed. The string's hash can later be retrieved from the SV with the "SvSHARED_HASH" macro. The idea here is that as the string table is used for shared hash keys these strings will have "SvPVX_const == HeKEY" and hash lookup will avoid string compare.

```
SV*  newSVpvn_share(const char* s, I32 len, U32 hash)
```

"newSVpvn_utf8"

Creates a new SV and copies a string (which may contain "NUL" ("\0") characters) into it. If "utf8" is true, calls "SvUTF8_on" on the new SV. Implemented as a wrapper around "newSVpvn_flags".

```
SV*  newSVpvn_utf8(const char* s, STRLEN len, U32 utf8)
```

"newSVpvs"

Like "newSVpvn", but takes a literal string instead of a string/length pair.

```
SV*  newSVpvs("literal string")
```

"newSVpvs_flags"

Like "newSVpvn_flags", but takes a literal string instead of a string/length pair.

```
SV*  newSVpvs_flags("literal string", U32 flags)
```

"newSVpv_share"

Like "newSVpvn_share", but takes a "NUL"-terminated string instead of a string/length pair.

```
SV*  newSVpv_share(const char* s, U32 hash)
```

"newSVpvs_share"

Like "newSVpvn_share", but takes a literal string instead of a string/length pair and omits the hash parameter.

```
SV*  newSVpvs_share("literal string")
```

"newSVrv"

Creates a new SV for the existing RV, "rv", to point to. If "rv" is not an RV then it will be upgraded to one. If "classname" is non-null then the new SV will be blessed in the specified package. The new SV is returned and its reference count is 1. The reference count 1 is owned by "rv". See also newRV_inc() and newRV_noinc() for creating a new RV properly.

```
SV*  newSVrv(SV *const rv, const char *const classname)
```

"newSVsv"

"newSVsv_nomg"

"newSVsv_flags"

These create a new SV which is an exact duplicate of the original SV (using "sv_setsv".)

They differ only in that "newSVsv" performs 'get' magic; "newSVsv_nomg" skips any magic; and "newSVsv_flags" allows you to explicitly set a "flags" parameter.

```
SV*  newSVsv      (SV *const old)
SV*  newSVsv_nomg (SV *const old)
SV*  newSVsv_flags(SV *const old, I32 flags)
```

"newSV_type"

Creates a new SV, of the type specified. The reference count for the new SV is set to 1.

```
SV*  newSV_type(const svtype type)
```

"newSVuv"

Creates a new SV and copies an unsigned integer into it. The reference count for the SV is set to 1.

```
SV*  newSVuv(const UV u)
```

"Nullsv"

Null SV pointer. (No longer available when "PERL_CORE" is defined.)

"PL_na"

A convenience variable which is typically used with "SvPV" when one doesn't care about the length of the string. It is usually more efficient to either declare a local

variable and use that instead or to use the "SvPV_nolen" macro.

```
 STRLEN  PL_na
```

"PL_sv_no"

This is the "false" SV. It is readonly. See "PL_sv_yes". Always refer to this as &PL_sv_no.

```
 SV  PL_sv_no
```

"PL_sv_undef"

This is the "undef" SV. It is readonly. Always refer to this as &PL_sv_undef.

```
 SV  PL_sv_undef
```

"PL_sv_yes"

This is the "true" SV. It is readonly. See "PL_sv_no". Always refer to this as &PL_sv_yes.

```
 SV  PL_sv_yes
```

"PL_sv_zero"

This readonly SV has a zero numeric value and a "0" string value. It's similar to "PL_sv_no" except for its string value. Can be used as a cheap alternative to mXPUSHi(0) for example. Always refer to this as &PL_sv_zero. Introduced in 5.28.

```
 SV  PL_sv_zero
```

"SAVE_DEFSV"

Localize $_. See "Localizing changes" in perlguts.

```
 void  SAVE_DEFSV
```

"sortsv"

In-place sort an array of SV pointers with the given comparison routine.

Currently this always uses mergesort. See "sortsv_flags" for a more flexible routine.

```
 void  sortsv(SV** array, size_t num_elts, SVCOMPARE_t cmp)
```

"sortsv_flags"

In-place sort an array of SV pointers with the given comparison routine, with various SORTf_* flag options.

```
 void  sortsv_flags(SV** array, size_t num_elts, SVCOMPARE_t cmp,
                    U32 flags)
```

"SV"

Described in perlguts.

"sv_2cv"

Using various gambits, try to get a CV from an SV; in addition, try if possible to set *st and *gvp to the stash and GV associated with it. The flags in "lref" are passed to "gv_fetchsv".

```
CV*  sv_2cv(SV* sv, HV **const st, GV **const gvp, const I32 lref)
```

"sv_2io"

Using various gambits, try to get an IO from an SV: the IO slot if its a GV; or the recursive result if we're an RV; or the IO slot of the symbol named after the PV if we're a string.

'Get' magic is ignored on the "sv" passed in, but will be called on "SvRV(sv)" if "sv" is an RV.

```
IO*  sv_2io(SV *const sv)
```

"sv_2iv_flags"

Return the integer value of an SV, doing any necessary string conversion. If "flags" has the "SV_GMAGIC" bit set, does an "mg_get()" first. Normally used via the "SvIV(sv)" and "SvIVx(sv)" macros.

```
IV  sv_2iv_flags(SV *const sv, const I32 flags)
```

"sv_2mortal"

Marks an existing SV as mortal. The SV will be destroyed "soon", either by an explicit call to "FREETMPS", or by an implicit call at places such as statement boundaries. "SvTEMP()" is turned on which means that the SV's string buffer can be "stolen" if this SV is copied. See also "sv_newmortal" and "sv_mortalcopy".

```
SV*  sv_2mortal(SV *const sv)
```

"sv_2nv_flags"

Return the num value of an SV, doing any necessary string or integer conversion. If "flags" has the "SV_GMAGIC" bit set, does an "mg_get()" first. Normally used via the "SvNV(sv)" and "SvNVx(sv)" macros.

```
NV  sv_2nv_flags(SV *const sv, const I32 flags)
```

"sv_2pvbyte"

Returns a pointer to the byte-encoded representation of the SV, and set *lp to its length. If the SV is marked as being encoded as UTF-8, it will downgrade it to a byte string as a side-effect, if possible. If the SV cannot be downgraded, this croaks.

Processes 'get' magic.

Usually accessed via the "SvPVbyte" macro.

char* sv_2pvbyte(SV *sv, STRLEN *const lp)

"sv_2pvutf8"

Return a pointer to the UTF-8-encoded representation of the SV, and set *lp to its length. May cause the SV to be upgraded to UTF-8 as a side-effect.

Usually accessed via the "SvPVutf8" macro.

char* sv_2pvutf8(SV *sv, STRLEN *const lp)

"sv_2uv_flags"

Return the unsigned integer value of an SV, doing any necessary string conversion. If "flags" has the "SV_GMAGIC" bit set, does an "mg_get()" first. Normally used via the "SvUV(sv)" and "SvUVx(sv)" macros.

UV sv_2uv_flags(SV *const sv, const I32 flags)

"sv_backoff"

Remove any string offset. You should normally use the "SvOOK_off" macro wrapper instead.

void sv_backoff(SV *const sv)

"sv_bless"

Blesses an SV into a specified package. The SV must be an RV. The package must be designated by its stash (see "gv_stashpv"). The reference count of the SV is unaffected.

SV* sv_bless(SV *const sv, HV *const stash)

"sv_catpv"

"sv_catpv_flags"

"sv_catpv_mg"

"sv_catpv_nomg"

These concatenate the "NUL"-terminated string "sstr" onto the end of the string which is in the SV. If the SV has the UTF-8 status set, then the bytes appended should be valid UTF-8.

They differ only in how they handle magic:

"sv_catpv_mg" performs both 'get' and 'set' magic.

"sv_catpv" performs only 'get' magic.

"sv_catpv_nomg" skips all magic.

"sv_catpv_flags" has an extra "flags" parameter which allows you to specify any combination of magic handling (using "SV_GMAGIC" and/or "SV_SMAGIC"), and to also

override the UTF-8 handling. By supplying the "SV_CATUTF8" flag, the appended string is forced to be interpreted as UTF-8; by supplying instead the "SV_CATBYTES" flag, it will be interpreted as just bytes. Either the SV or the string appended will be upgraded to UTF-8 if necessary.

```
 void  sv_catpv      (SV *const dsv, const char* sstr)
 void  sv_catpv_flags(SV *dsv, const char *sstr, const I32 flags)
 void  sv_catpv_mg   (SV *const dsv, const char *const sstr)
 void  sv_catpv_nomg (SV *const dsv, const char* sstr)
```

"sv_catpvf"

"sv_catpvf_nocontext"

"sv_catpvf_mg"

"sv_catpvf_mg_nocontext"

These process their arguments like "sprintf", and append the formatted output to an SV. As with "sv_vcatpvfn", argument reordering is not supporte when called with a non-null C-style variable argument list.

If the appended data contains "wide" characters (including, but not limited to, SVs with a UTF-8 PV formatted with %s, and characters >255 formatted with %c), the original SV might get upgraded to UTF-8.

If the original SV was UTF-8, the pattern should be valid UTF-8; if the original SV was bytes, the pattern should be too.

All perform 'get' magic, but only "sv_catpvf_mg" and "sv_catpvf_mg_nocontext" perform 'set' magic.

"sv_catpvf_nocontext" and "sv_catpvf_mg_nocontext" do not take a thread context ("aTHX") parameter, so are used in situations where the caller doesn't already have the thread context.

NOTE: "sv_catpvf" must be explicitly called as "Perl_sv_catpvf" with an "aTHX_" parameter.

NOTE: "sv_catpvf_mg" must be explicitly called as "Perl_sv_catpvf_mg" with an "aTHX_" parameter.

```
 void  Perl_sv_catpvf       (pTHX_ SV *const sv,
                             const char *const pat, ...)
 void  sv_catpvf_nocontext   (SV *const sv, const char *const pat,
                             ...)
```

```
    void  Perl_sv_catpvf_mg     (pTHX_ SV *const sv,
                                 const char *const pat, ...)
    void  sv_catpvf_mg_nocontext(SV *const sv, const char *const pat,
                                 ...)
```

"sv_catpvn"

"sv_catpvn_flags"

"sv_catpvn_mg"

"sv_catpvn_nomg"

These concatenate the "len" bytes of the string beginning at "ptr" onto the end of the string which is in "dsv". The caller must make sure "ptr" contains at least "len" bytes.

For all but "sv_catpvn_flags", the string appended is assumed to be valid UTF-8 if the SV has the UTF-8 status set, and a string of bytes otherwise.

They differ in that:

"sv_catpvn_mg" performs both 'get' and 'set' magic on "dsv".

"sv_catpvn" performs only 'get' magic.

"sv_catpvn_nomg" skips all magic.

"sv_catpvn_flags" has an extra "flags" parameter which allows you to specify any combination of magic handling (using "SV_GMAGIC" and/or "SV_SMAGIC") and to also override the UTF-8 handling. By supplying the "SV_CATBYTES" flag, the appended string is interpreted as plain bytes; by supplying instead the "SV_CATUTF8" flag, it will be interpreted as UTF-8, and the "dsv" will be upgraded to UTF-8 if necessary.

"sv_catpvn", "sv_catpvn_mg", and "sv_catpvn_nomg" are implemented in terms of "sv_catpvn_flags".

```
    void  sv_catpvn      (SV *dsv, const char *sstr, STRLEN len)
    void  sv_catpvn_flags(SV *const dsv, const char *sstr,
                          const STRLEN len, const I32 flags)
    void  sv_catpvn_mg   (SV *dsv, const char *sstr, STRLEN len)
    void  sv_catpvn_nomg (SV *dsv, const char *sstr, STRLEN len)
```

"sv_catpvs"

Like "sv_catpvn", but takes a literal string instead of a string/length pair.

```
    void  sv_catpvs(SV* sv, "literal string")
```

"sv_catpvs_flags"

Like "sv_catpvn_flags", but takes a literal string instead of a string/length pair.

```
 void  sv_catpvs_flags(SV* sv, "literal string", I32 flags)
```

"sv_catpvs_mg"

Like "sv_catpvn_mg", but takes a literal string instead of a string/length pair.

```
 void  sv_catpvs_mg(SV* sv, "literal string")
```

"sv_catpvs_nomg"

Like "sv_catpvn_nomg", but takes a literal string instead of a string/length pair.

```
 void  sv_catpvs_nomg(SV* sv, "literal string")
```

"sv_catsv"

"sv_catsv_flags"

"sv_catsv_mg"

"sv_catsv_nomg"

These concatenate the string from SV "sstr" onto the end of the string in SV "dsv".

If "sstr" is null, these are no-ops; otherwise only "dsv" is modified.

They differ only in what magic they perform:

"sv_catsv_mg" performs 'get' magic on both SVs before the copy, and 'set' magic on "dsv" afterwards.

"sv_catsv" performs just 'get' magic, on both SVs.

"sv_catsv_nomg" skips all magic.

"sv_catsv_flags" has an extra "flags" parameter which allows you to use "SV_GMAGIC" and/or "SV_SMAGIC" to specify any combination of magic handling (although either both or neither SV will have 'get' magic applied to it.)

"sv_catsv", "sv_catsv_mg", and "sv_catsv_nomg" are implemented in terms of "sv_catsv_flags".

```
 void  sv_catsv      (SV *dsv, SV *sstr)
 void  sv_catsv_flags(SV *const dsv, SV *const sstr,
                      const I32 flags)
 void  sv_catsv_mg   (SV *dsv, SV *sstr)
 void  sv_catsv_nomg (SV *dsv, SV *sstr)
```

"sv_chop"

Efficient removal of characters from the beginning of the string buffer. "SvPOK(sv)", or at least "SvPOKp(sv)", must be true and "ptr" must be a pointer to somewhere inside the string buffer. "ptr" becomes the first character of the adjusted string. Uses

the "OOK" hack. On return, only "SvPOK(sv)" and "SvPOKp(sv)" among the "OK" flags will be true.

Beware: after this function returns, "ptr" and SvPVX_const(sv) may no longer refer to the same chunk of data.

The unfortunate similarity of this function's name to that of Perl's "chop" operator is strictly coincidental. This function works from the left; "chop" works from the right.

```
 void  sv_chop(SV *const sv, const char *const ptr)
```

"sv_clear"

Clear an SV: call any destructors, free up any memory used by the body, and free the body itself. The SV's head is not freed, although its type is set to all 1's so that it won't inadvertently be assumed to be live during global destruction etc. This function should only be called when "REFCNT" is zero. Most of the time you'll want to call "sv_free()" (or its macro wrapper "SvREFCNT_dec") instead.

```
 void  sv_clear(SV *const orig_sv)
```

"sv_cmp"

Compares the strings in two SVs. Returns -1, 0, or 1 indicating whether the string in "sv1" is less than, equal to, or greater than the string in "sv2". Is UTF-8 and 'use?bytes' aware, handles get magic, and will coerce its args to strings if necessary. See also "sv_cmp_locale".

```
 I32  sv_cmp(SV *const sv1, SV *const sv2)
```

"sv_cmp_flags"

Compares the strings in two SVs. Returns -1, 0, or 1 indicating whether the string in "sv1" is less than, equal to, or greater than the string in "sv2". Is UTF-8 and 'use?bytes' aware and will coerce its args to strings if necessary. If the flags has the "SV_GMAGIC" bit set, it handles get magic. See also "sv_cmp_locale_flags".

```
 I32  sv_cmp_flags(SV *const sv1, SV *const sv2, const U32 flags)
```

"sv_cmp_locale"

Compares the strings in two SVs in a locale-aware manner. Is UTF-8 and 'use?bytes' aware, handles get magic, and will coerce its args to strings if necessary. See also "sv_cmp".

```
 I32  sv_cmp_locale(SV *const sv1, SV *const sv2)
```

"sv_cmp_locale_flags"

Compares the strings in two SVs in a locale-aware manner. Is UTF-8 and 'use?bytes' aware and will coerce its args to strings if necessary. If the flags contain "SV_GMAGIC", it handles get magic. See also "sv_cmp_flags".

```
I32  sv_cmp_locale_flags(SV *const sv1, SV *const sv2,
                         const U32 flags)
```

"sv_collxfrm"

This calls "sv_collxfrm_flags" with the SV_GMAGIC flag. See "sv_collxfrm_flags".

```
char*  sv_collxfrm(SV *const sv, STRLEN *const nxp)
```

"sv_collxfrm_flags"

Add Collate Transform magic to an SV if it doesn't already have it. If the flags contain "SV_GMAGIC", it handles get-magic.

Any scalar variable may carry "PERL_MAGIC_collxfrm" magic that contains the scalar data of the variable, but transformed to such a format that a normal memory comparison can be used to compare the data according to the locale settings.

```
char*  sv_collxfrm_flags(SV *const sv, STRLEN *const nxp,
                         I32 const flags)
```

"sv_copypv"

"sv_copypv_nomg"

"sv_copypv_flags"

These copy a stringified representation of the source SV into the destination SV. They automatically perform coercion of numeric values into strings. Guaranteed to preserve the "UTF8" flag even from overloaded objects. Similar in nature to "sv_2pv[_flags]" but they operate directly on an SV instead of just the string. Mostly they use ""sv_2pv_flags"" in perlintern to do the work, except when that would lose the UTF-8'ness of the PV.

The three forms differ only in whether or not they perform 'get magic' on "sv". "sv_copypv_nomg" skips 'get magic'; "sv_copypv" performs it; and "sv_copypv_flags" either performs it (if the "SV_GMAGIC" bit is set in "flags") or doesn't (if that bit is cleared).

```
void  sv_copypv      (SV *const dsv, SV *const ssv)
void  sv_copypv_nomg (SV *const dsv, SV *const ssv)
void  sv_copypv_flags(SV *const dsv, SV *const ssv,
                      const I32 flags)
```

"SvCUR"

Returns the length, in bytes, of the PV inside the SV. Note that this may not match Perl's "length"; for that, use "sv_len_utf8(sv)". See "SvLEN" also.

```
STRLEN  SvCUR(SV* sv)
```

"SvCUR_set"

Sets the current length, in bytes, of the C string which is in the SV. See "SvCUR" and "SvIV_set">.

```
void  SvCUR_set(SV* sv, STRLEN len)
```

"sv_dec"

"sv_dec_nomg"

These auto-decrement the value in the SV, doing string to numeric conversion if necessary. They both handle operator overloading.

They differ only in that:

"sv_dec" handles 'get' magic; "sv_dec_nomg" skips 'get' magic.

```
void  sv_dec(SV *const sv)
```

"sv_derived_from"

Exactly like "sv_derived_from_pv", but doesn't take a "flags" parameter.

```
bool  sv_derived_from(SV* sv, const char *const name)
```

"sv_derived_from_pv"

Exactly like "sv_derived_from_pvn", but takes a nul-terminated string instead of a string/length pair.

```
bool  sv_derived_from_pv(SV* sv, const char *const name,
                         U32 flags)
```

"sv_derived_from_pvn"

Returns a boolean indicating whether the SV is derived from the specified class at the C level. To check derivation at the Perl level, call "isa()" as a normal Perl method.

Currently, the only significant value for "flags" is SVf_UTF8.

```
bool  sv_derived_from_pvn(SV* sv, const char *const name,
                          const STRLEN len, U32 flags)
```

"sv_derived_from_sv"

Exactly like "sv_derived_from_pvn", but takes the name string in the form of an SV instead of a string/length pair. This is the advised form.

```
bool  sv_derived_from_sv(SV* sv, SV *namesv, U32 flags)
```

"sv_does"

Like "sv_does_pv", but doesn't take a "flags" parameter.

```
 bool  sv_does(SV* sv, const char *const name)
```

"sv_does_pv"

Like "sv_does_sv", but takes a nul-terminated string instead of an SV.

```
 bool  sv_does_pv(SV* sv, const char *const name, U32 flags)
```

"sv_does_pvn"

Like "sv_does_sv", but takes a string/length pair instead of an SV.

```
 bool  sv_does_pvn(SV* sv, const char *const name,
                   const STRLEN len, U32 flags)
```

"sv_does_sv"

Returns a boolean indicating whether the SV performs a specific, named role. The SV can be a Perl object or the name of a Perl class.

```
 bool  sv_does_sv(SV* sv, SV* namesv, U32 flags)
```

"SvEND"

Returns a pointer to the spot just after the last character in the string which is in the SV, where there is usually a trailing "NUL" character (even though Perl scalars do not strictly require it). See "SvCUR". Access the character as "*(SvEND(sv))".

Warning: If "SvCUR" is equal to "SvLEN", then "SvEND" points to unallocated memory.

```
 char*  SvEND(SV* sv)
```

"sv_eq"

Returns a boolean indicating whether the strings in the two SVs are identical. Is UTF-8 and 'use?bytes' aware, handles get magic, and will coerce its args to strings if necessary.

```
 I32  sv_eq(SV* sv1, SV* sv2)
```

"sv_eq_flags"

Returns a boolean indicating whether the strings in the two SVs are identical. Is UTF-8 and 'use?bytes' aware and coerces its args to strings if necessary. If the flags has the "SV_GMAGIC" bit set, it handles get-magic, too.

```
 I32  sv_eq_flags(SV* sv1, SV* sv2, const U32 flags)
```

"sv_force_normal"

Undo various types of fakery on an SV: if the PV is a shared string, make a private copy; if we're a ref, stop refing; if we're a glob, downgrade to an "xpvmg". See also

"sv_force_normal_flags".

```
void  sv_force_normal(SV *sv)
```

"sv_force_normal_flags"

Undo various types of fakery on an SV, where fakery means "more than" a string: if the PV is a shared string, make a private copy; if we're a ref, stop refing; if we're a glob, downgrade to an "xpvmg"; if we're a copy-on-write scalar, this is the on-write time when we do the copy, and is also used locally; if this is a vstring, drop the vstring magic. If "SV_COW_DROP_PV" is set then a copy-on-write scalar drops its PV buffer (if any) and becomes "SvPOK_off" rather than making a copy. (Used where this scalar is about to be set to some other value.) In addition, the "flags" parameter gets passed to "sv_unref_flags()" when unreffing. "sv_force_normal" calls this function with flags set to 0.

This function is expected to be used to signal to perl that this SV is about to be written to, and any extra book-keeping needs to be taken care of. Hence, it croaks on read-only values.

```
void  sv_force_normal_flags(SV *const sv, const U32 flags)
```

"sv_free"

Decrement an SV's reference count, and if it drops to zero, call "sv_clear" to invoke destructors and free up any memory used by the body; finally, deallocating the SV's head itself. Normally called via a wrapper macro "SvREFCNT_dec".

```
void  sv_free(SV *const sv)
```

"SvGAMAGIC"

Returns true if the SV has get magic or overloading. If either is true then the scalar is active data, and has the potential to return a new value every time it is accessed. Hence you must be careful to only read it once per user logical operation and work with that returned value. If neither is true then the scalar's value cannot change unless written to.

```
U32  SvGAMAGIC(SV* sv)
```

"SvGETMAGIC"

Invokes "mg_get" on an SV if it has 'get' magic. For example, this will call "FETCH" on a tied variable. This macro evaluates its argument more than once.

```
void  SvGETMAGIC(SV* sv)
```

"sv_gets"

Get a line from the filehandle and store it into the SV, optionally appending to the currently-stored string. If "append" is not 0, the line is appended to the SV instead of overwriting it. "append" should be set to the byte offset that the appended string should start at in the SV (typically, "SvCUR(sv)" is a suitable choice).

```
char*  sv_gets(SV *const sv, PerlIO *const fp, I32 append)
```

"sv_get_backrefs"

NOTE: "sv_get_backrefs" is experimental and may change or be removed without notice.

If "sv" is the target of a weak reference then it returns the back references structure associated with the sv; otherwise return "NULL".

When returning a non-null result the type of the return is relevant. If it is an AV then the elements of the AV are the weak reference RVs which point at this item. If it is any other type then the item itself is the weak reference.

See also "Perl_sv_add_backref()", "Perl_sv_del_backref()", "Perl_sv_kill_backrefs()"

```
SV*  sv_get_backrefs(SV *const sv)
```

"SvGROW"

Expands the character buffer in the SV so that it has room for the indicated number of bytes (remember to reserve space for an extra trailing "NUL" character). Calls "sv_grow" to perform the expansion if necessary. Returns a pointer to the character buffer. SV must be of type >= "SVt_PV". One alternative is to call "sv_grow" if you are not sure of the type of SV.

You might mistakenly think that "len" is the number of bytes to add to the existing size, but instead it is the total size "sv" should be.

```
char *  SvGROW(SV* sv, STRLEN len)
```

"sv_inc"

"sv_inc_nomg"

These auto-increment the value in the SV, doing string to numeric conversion if necessary. They both handle operator overloading.

They differ only in that "sv_inc" performs 'get' magic; "sv_inc_nomg" skips any magic.

```
void  sv_inc(SV *const sv)
```

"sv_insert"

Inserts and/or replaces a string at the specified offset/length within the SV. Similar to the Perl "substr()" function, with "littlelen" bytes starting at "little" replacing "len" bytes of the string in "bigstr" starting at "offset". Handles get

magic.

```
 void  sv_insert(SV *const bigstr, const STRLEN offset,
                 const STRLEN len, const char *const little,
                 const STRLEN littlelen)
```

"sv_insert_flags"

Same as "sv_insert", but the extra "flags" are passed to the "SvPV_force_flags" that applies to "bigstr".

```
 void  sv_insert_flags(SV *const bigstr, const STRLEN offset,
                       const STRLEN len, const char *little,
                       const STRLEN littlelen, const U32 flags)
```

"SvIOK"

Returns a U32 value indicating whether the SV contains an integer.

```
 U32  SvIOK(SV* sv)
```

"SvIOK_notUV"

Returns a boolean indicating whether the SV contains a signed integer.

```
 bool  SvIOK_notUV(SV* sv)
```

"SvIOK_off"

Unsets the IV status of an SV.

```
 void  SvIOK_off(SV* sv)
```

"SvIOK_on"

Tells an SV that it is an integer.

```
 void  SvIOK_on(SV* sv)
```

"SvIOK_only"

Tells an SV that it is an integer and disables all other "OK" bits.

```
 void  SvIOK_only(SV* sv)
```

"SvIOK_only_UV"

Tells an SV that it is an unsigned integer and disables all other "OK" bits.

```
 void  SvIOK_only_UV(SV* sv)
```

"SvIOKp"

Returns a U32 value indicating whether the SV contains an integer. Checks the private setting. Use "SvIOK" instead.

```
 U32  SvIOKp(SV* sv)
```

"SvIOK_UV"

Returns a boolean indicating whether the SV contains an integer that must be interpreted as unsigned. A non-negative integer whose value is within the range of both an IV and a UV may be flagged as either "SvUOK" or "SvIOK".

```
bool  SvIOK_UV(SV* sv)
```

"sv_isa"

Returns a boolean indicating whether the SV is blessed into the specified class. This does not check for subtypes or method overloading. Use "sv_isa_sv" to verify an inheritance relationship in the same way as the "isa" operator by respecting any "isa()" method overloading; or "sv_derived_from_sv" to test directly on the actual object type.

```
int  sv_isa(SV* sv, const char *const name)
```

"sv_isa_sv"

NOTE: "sv_isa_sv" is experimental and may change or be removed without notice.

Returns a boolean indicating whether the SV is an object reference and is derived from the specified class, respecting any "isa()" method overloading it may have. Returns false if "sv" is not a reference to an object, or is not derived from the specified class.

This is the function used to implement the behaviour of the "isa" operator.

Does not invoke magic on "sv".

Not to be confused with the older "sv_isa" function, which does not use an overloaded "isa()" method, nor will check subclassing.

```
bool  sv_isa_sv(SV* sv, SV* namesv)
```

"SvIsCOW"

Returns a U32 value indicating whether the SV is Copy-On-Write (either shared hash key scalars, or full Copy On Write scalars if 5.9.0 is configured for COW).

```
U32  SvIsCOW(SV* sv)
```

"SvIsCOW_shared_hash"

Returns a boolean indicating whether the SV is Copy-On-Write shared hash key scalar.

```
bool  SvIsCOW_shared_hash(SV* sv)
```

"sv_isobject"

Returns a boolean indicating whether the SV is an RV pointing to a blessed object. If the SV is not an RV, or if the object is not blessed, then this will return false.

```
int  sv_isobject(SV* sv)
```

"SvIV"

"SvIVx"

"SvIV_nomg"

These coerce the given SV to IV and return it. The returned value in many circumstances will get stored in "sv"'s IV slot, but not in all cases. (Use "sv_setiv" to make sure it does).

"SvIVx" is different from the others in that it is guaranteed to evaluate "sv" exactly once; the others may evaluate it multiple times. Only use this form if "sv" is an expression with side effects, otherwise use the more efficient "SvIV".

"SvIV_nomg" is the same as "SvIV", but does not perform 'get' magic.

```
IV  SvIV(SV* sv)
```

"SvIV_set"

Set the value of the IV pointer in sv to val. It is possible to perform the same function of this macro with an lvalue assignment to "SvIVX". With future Perls, however, it will be more efficient to use "SvIV_set" instead of the lvalue assignment to "SvIVX".

```
void  SvIV_set(SV* sv, IV val)
```

"SvIVX"

Returns the raw value in the SV's IV slot, without checks or conversions. Only use when you are sure "SvIOK" is true. See also "SvIV".

```
IV  SvIVX(SV* sv)
```

"SvLEN"

Returns the size of the string buffer in the SV, not including any part attributable to "SvOOK". See "SvCUR".

```
STRLEN  SvLEN(SV* sv)
```

"sv_len"

Returns the length of the string in the SV. Handles magic and type coercion and sets the UTF8 flag appropriately. See also "SvCUR", which gives raw access to the "xpv_cur" slot.

```
STRLEN  sv_len(SV *const sv)
```

"SvLEN_set"

Set the size of the string buffer for the SV. See "SvLEN".

```
void  SvLEN_set(SV* sv, STRLEN len)
```

"sv_len_utf8"

Returns the number of characters in the string in an SV, counting wide UTF-8 bytes as a single character. Handles magic and type coercion.

```
STRLEN  sv_len_utf8(SV *const sv)
```

"SvLOCK"

Arranges for a mutual exclusion lock to be obtained on "sv" if a suitable module has been loaded.

```
void  SvLOCK(SV* sv)
```

"sv_magic"

Adds magic to an SV. First upgrades "sv" to type "SVt_PVMG" if necessary, then adds a new magic item of type "how" to the head of the magic list.

See "sv_magicext" (which "sv_magic" now calls) for a description of the handling of the "name" and "namlen" arguments.

You need to use "sv_magicext" to add magic to "SvREADONLY" SVs and also to add more than one instance of the same "how".

```
void  sv_magic(SV *const sv, SV *const obj, const int how,
               const char *const name, const I32 namlen)
```

"sv_magicext"

Adds magic to an SV, upgrading it if necessary. Applies the supplied "vtable" and returns a pointer to the magic added.

Note that "sv_magicext" will allow things that "sv_magic" will not. In particular, you can add magic to "SvREADONLY" SVs, and add more than one instance of the same "how".

If "namlen" is greater than zero then a "savepvn" copy of "name" is stored, if "namlen" is zero then "name" is stored as-is and - as another special case - if "(name && namlen == HEf_SVKEY)" then "name" is assumed to contain an SV* and is stored as-is with its "REFCNT" incremented.

(This is now used as a subroutine by "sv_magic".)

```
MAGIC *  sv_magicext(SV *const sv, SV *const obj, const int how,
                     const MGVTBL *const vtbl,
                     const char *const name, const I32 namlen)
```

"SvMAGIC_set"

Set the value of the MAGIC pointer in "sv" to val. See "SvIV_set".

```
 void  SvMAGIC_set(SV* sv, MAGIC* val)
```

"sv_mortalcopy"

Creates a new SV which is a copy of the original SV (using "sv_setsv"). The new SV is marked as mortal. It will be destroyed "soon", either by an explicit call to "FREETMPS", or by an implicit call at places such as statement boundaries. See also "sv_newmortal" and "sv_2mortal".

```
 SV*  sv_mortalcopy(SV *const oldsv)
```

"sv_mortalcopy_flags"

Like "sv_mortalcopy", but the extra "flags" are passed to the "sv_setsv_flags".

```
 SV*  sv_mortalcopy_flags(SV *const oldsv, U32 flags)
```

"sv_newmortal"

Creates a new null SV which is mortal. The reference count of the SV is set to 1. It will be destroyed "soon", either by an explicit call to "FREETMPS", or by an implicit call at places such as statement boundaries. See also "sv_mortalcopy" and "sv_2mortal".

```
 SV*  sv_newmortal()
```

"SvNIOK"

Returns a U32 value indicating whether the SV contains a number, integer or double.

```
 U32  SvNIOK(SV* sv)
```

"SvNIOK_off"

Unsets the NV/IV status of an SV.

```
 void  SvNIOK_off(SV* sv)
```

"SvNIOKp"

Returns a U32 value indicating whether the SV contains a number, integer or double. Checks the private setting. Use "SvNIOK" instead.

```
 U32  SvNIOKp(SV* sv)
```

"SvNOK"

Returns a U32 value indicating whether the SV contains a double.

```
 U32  SvNOK(SV* sv)
```

"SvNOK_off"

Unsets the NV status of an SV.

```
 void  SvNOK_off(SV* sv)
```

"SvNOK_on"

Tells an SV that it is a double.

```
void  SvNOK_on(SV* sv)
```

"SvNOK_only"

Tells an SV that it is a double and disables all other OK bits.

```
void  SvNOK_only(SV* sv)
```

"SvNOKp"

Returns a U32 value indicating whether the SV contains a double. Checks the private setting. Use "SvNOK" instead.

```
U32  SvNOKp(SV* sv)
```

"sv_nolocking"

"DEPRECATED!" It is planned to remove "sv_nolocking" from a future release of Perl. Do not use it for new code; remove it from existing code.

Dummy routine which "locks" an SV when there is no locking module present. Exists to avoid test for a "NULL" function pointer and because it could potentially warn under some level of strict-ness.

"Superseded" by "sv_nosharing()".

```
void  sv_nolocking(SV *sv)
```

"sv_nounlocking"

"DEPRECATED!" It is planned to remove "sv_nounlocking" from a future release of Perl. Do not use it for new code; remove it from existing code.

Dummy routine which "unlocks" an SV when there is no locking module present. Exists to avoid test for a "NULL" function pointer and because it could potentially warn under some level of strict-ness.

"Superseded" by "sv_nosharing()".

```
void  sv_nounlocking(SV *sv)
```

"SvNV"

"SvNVx"

"SvNV_nomg"

These coerce the given SV to NV and return it. The returned value in many circumstances will get stored in "sv"'s NV slot, but not in all cases. (Use "sv_setnv" to make sure it does).

"SvNVx" is different from the others in that it is guaranteed to evaluate "sv" exactly once; the others may evaluate it multiple times. Only use this form if "sv" is an

expression with side effects, otherwise use the more efficient "SvNV".

"SvNV_nomg" is the same as "SvNV", but does not perform 'get' magic.

```
 NV  SvNV(SV* sv)
```

"SvNV_set"

Set the value of the NV pointer in "sv" to val. See "SvIV_set".

```
 void  SvNV_set(SV* sv, NV val)
```

"SvNVX"

Returns the raw value in the SV's NV slot, without checks or conversions. Only use when you are sure "SvNOK" is true. See also "SvNV".

```
 NV  SvNVX(SV* sv)
```

"SvOK"

Returns a U32 value indicating whether the value is defined. This is only meaningful for scalars.

```
 U32  SvOK(SV* sv)
```

"SvOOK"

Returns a U32 indicating whether the pointer to the string buffer is offset. This hack is used internally to speed up removal of characters from the beginning of a "SvPV". When "SvOOK" is true, then the start of the allocated string buffer is actually "SvOOK_offset()" bytes before "SvPVX". This offset used to be stored in "SvIVX", but is now stored within the spare part of the buffer.

```
 U32  SvOOK(SV* sv)
```

"SvOOK_off"

Remove any string offset.

```
 void  SvOOK_off(SV * sv)
```

"SvOOK_offset"

Reads into "len" the offset from "SvPVX" back to the true start of the allocated buffer, which will be non-zero if "sv_chop" has been used to efficiently remove characters from start of the buffer. Implemented as a macro, which takes the address of "len", which must be of type "STRLEN". Evaluates "sv" more than once. Sets "len" to 0 if "SvOOK(sv)" is false.

```
 void  SvOOK_offset(SV*sv, STRLEN len)
```

"SvPOK"

Returns a U32 value indicating whether the SV contains a character string.

U32 SvPOK(SV* sv)

"SvPOK_off"

Unsets the PV status of an SV.

void SvPOK_off(SV* sv)

"SvPOK_on"

Tells an SV that it is a string.

void SvPOK_on(SV* sv)

"SvPOK_only"

Tells an SV that it is a string and disables all other "OK" bits. Will also turn off the UTF-8 status.

void SvPOK_only(SV* sv)

"SvPOK_only_UTF8"

Tells an SV that it is a string and disables all other "OK" bits, and leaves the UTF-8 status as it was.

void SvPOK_only_UTF8(SV* sv)

"SvPOKp"

Returns a U32 value indicating whether the SV contains a character string. Checks the private setting. Use "SvPOK" instead.

U32 SvPOKp(SV* sv)

"sv_pos_b2u"

Converts the value pointed to by "offsetp" from a count of bytes from the start of the string, to a count of the equivalent number of UTF-8 chars. Handles magic and type coercion.

Use "sv_pos_b2u_flags" in preference, which correctly handles strings longer than 2Gb.

void sv_pos_b2u(SV *const sv, I32 *const offsetp)

"sv_pos_b2u_flags"

Converts "offset" from a count of bytes from the start of the string, to a count of the equivalent number of UTF-8 chars. Handles type coercion. "flags" is passed to "SvPV_flags", and usually should be "SV_GMAGIC|SV_CONST_RETURN" to handle magic.

STRLEN sv_pos_b2u_flags(SV *const sv, STRLEN const offset,
U32 flags)

"sv_pos_u2b"

Converts the value pointed to by "offsetp" from a count of UTF-8 chars from the start

of the string, to a count of the equivalent number of bytes; if "lenp" is non-zero, it does the same to "lenp", but this time starting from the offset, rather than from the start of the string. Handles magic and type coercion.

Use "sv_pos_u2b_flags" in preference, which correctly handles strings longer than 2Gb.

```
void  sv_pos_u2b(SV *const sv, I32 *const offsetp,
                 I32 *const lenp)
```

"sv_pos_u2b_flags"

Converts the offset from a count of UTF-8 chars from the start of the string, to a count of the equivalent number of bytes; if "lenp" is non-zero, it does the same to "lenp", but this time starting from "offset", rather than from the start of the string. Handles type coercion. "flags" is passed to "SvPV_flags", and usually should be "SV_GMAGIC|SV_CONST_RETURN" to handle magic.

```
STRLEN  sv_pos_u2b_flags(SV *const sv, STRLEN uoffset,
                         STRLEN *const lenp, U32 flags)
```

"SvPV"

"SvPVx"

"SvPV_nomg"

"SvPV_nolen"

"SvPVx_nolen"

"SvPV_nomg_nolen"

"SvPV_mutable"

"SvPV_const"

"SvPVx_const"

"SvPV_nolen_const"

"SvPVx_nolen_const"

"SvPV_nomg_const"

"SvPV_nomg_const_nolen"

"SvPV_flags"

"SvPV_flags_const"

"SvPV_flags_mutable"

"SvPVbyte"

"SvPVbyte_nomg"

"SvPVbyte_nolen"

"SvPVbytex_nolen"

"SvPVbytex"

"SvPVbyte_or_null"

"SvPVbyte_or_null_nomg"

"SvPVutf8"

"SvPVutf8x"

"SvPVutf8_nomg"

"SvPVutf8_nolen"

"SvPVutf8_or_null"

"SvPVutf8_or_null_nomg"

All these return a pointer to the string in "sv", or a stringified form of "sv" if it does not contain a string. The SV may cache the stringified version becoming "SvPOK". This is a very basic and common operation, so there are lots of slightly different versions of it.

Note that there is no guarantee that the return value of "SvPV(sv)", for example, is equal to "SvPVX(sv)", or that "SvPVX(sv)" contains valid data, or that successive calls to "SvPV(sv)" (or another of these forms) will return the same pointer value each time. This is due to the way that things like overloading and Copy-On-Write are handled. In these cases, the return value may point to a temporary buffer or similar. If you absolutely need the "SvPVX" field to be valid (for example, if you intend to write to it), then see "SvPV_force".

The differences between the forms are:

The forms with neither "byte" nor "utf8" in their names (e.g., "SvPV" or "SvPV_nolen") can expose the SV's internal string buffer. If that buffer consists entirely of bytes 0-255 and includes any bytes above 127, then you MUST consult "SvUTF8" to determine the actual code points the string is meant to contain. Generally speaking, it is probably safer to prefer "SvPVbyte", "SvPVutf8", and the like. See "How do I pass a Perl string to a C library?" in perlguts for more details.

The forms with "flags" in their names allow you to use the "flags" parameter to specify to process 'get' magic (by setting the "SV_GMAGIC" flag) or to skip 'get' magic (by clearing it). The other forms process 'get' magic, except for the ones with "nomg" in their names, which skip 'get' magic.

The forms that take a "len" parameter will set that variable to the byte length of the

resultant string (these are macros, so don't use &len).

The forms with "nolen" in their names indicate they don't have a "len" parameter. They should be used only when it is known that the PV is a C string, terminated by a NUL byte, and without intermediate NUL characters; or when you don't care about its length.

The forms with "const" in their names return "const?char?*" so that the compiler will hopefully complain if you were to try to modify the contents of the string (unless you cast away const yourself).

The other forms return a mutable pointer so that the string is modifiable by the caller; this is emphasized for the ones with "mutable" in their names.

The forms whose name ends in "x" are the same as the corresponding form without the "x", but the "x" form is guaranteed to evaluate "sv" exactly once, with a slight loss of efficiency. Use this if "sv" is an expression with side effects.

"SvPVutf8" is like "SvPV", but converts "sv" to UTF-8 first if not already UTF-8.

Similiarly, the other forms with "utf8" in their names correspond to their respective forms without.

"SvPVutf8_or_null" and "SvPVutf8_or_null_nomg" don't have corresponding non-"utf8" forms. Instead they are like "SvPVutf8_nomg", but when "sv" is undef, they return "NULL".

"SvPVbyte" is like "SvPV", but converts "sv" to byte representation first if currently encoded as UTF-8. If "sv" cannot be downgraded from UTF-8, it croaks. Similiarly, the other forms with "byte" in their names correspond to their respective forms without.

"SvPVbyte_or_null" doesn't have a corresponding non-"byte" form. Instead it is like "SvPVbyte", but when "sv" is undef, it returns "NULL".

```
 char*        SvPV               (SV* sv, STRLEN len)
 char*        SvPVx              (SV* sv, STRLEN len)
 char*        SvPV_nomg          (SV* sv, STRLEN len)
 char*        SvPV_nolen         (SV* sv)
 char*        SvPVx_nolen        (SV* sv)
 char*        SvPV_nomg_nolen    (SV* sv)
 char*        SvPV_mutable       (SV* sv, STRLEN len)
 const char*  SvPV_const         (SV* sv, STRLEN len)
```

```
    const char*   SvPVx_const          (SV* sv, STRLEN len)
    const char*   SvPV_nolen_const     (SV* sv)
    const char*   SvPVx_nolen_const    (SV* sv)
    const char*   SvPV_nomg_const      (SV* sv, STRLEN len)
    const char*   SvPV_nomg_const_nolen(SV* sv)
    char *        SvPV_flags           (SV * sv, STRLEN len,
                                        U32 flags)
    const char *  SvPV_flags_const     (SV * sv, STRLEN len,
                                        U32 flags)
    char *        SvPV_flags_mutable   (SV * sv, STRLEN len,
                                        U32 flags)
    char*         SvPVbyte             (SV* sv, STRLEN len)
    char*         SvPVbyte_nomg        (SV* sv, STRLEN len)
    char*         SvPVbyte_nolen       (SV* sv)
    char*         SvPVbytex_nolen      (SV* sv)
    char*         SvPVbytex            (SV* sv, STRLEN len)
    char*         SvPVbyte_or_null     (SV* sv, STRLEN len)
    char*         SvPVbyte_or_null_nomg(SV* sv, STRLEN len)
    char*         SvPVutf8             (SV* sv, STRLEN len)
    char*         SvPVutf8x            (SV* sv, STRLEN len)
    char*         SvPVutf8_nomg        (SV* sv, STRLEN len)
    char*         SvPVutf8_nolen       (SV* sv)
    char*         SvPVutf8_or_null     (SV* sv, STRLEN len)
    char*         SvPVutf8_or_null_nomg(SV* sv, STRLEN len)
```

"SvPVbyte"

Like "SvPV", but converts "sv" to byte representation first if necessary. If the SV cannot be downgraded from UTF-8, this croaks.

```
    char*  SvPVbyte(SV* sv, STRLEN len)
```

"SvPVbyte_force"

Like "SvPV_force", but converts "sv" to byte representation first if necessary. If the SV cannot be downgraded from UTF-8, this croaks.

```
    char*  SvPVbyte_force(SV* sv, STRLEN len)
```

"SvPVbyte_nolen"

Like "SvPV_nolen", but converts "sv" to byte representation first if necessary. If the SV cannot be downgraded from UTF-8, this croaks.

```
 char*  SvPVbyte_nolen(SV* sv)
```

"SvPVbyte_nomg"

Like "SvPVbyte", but does not process get magic.

```
 char*  SvPVbyte_nomg(SV* sv, STRLEN len)
```

"SvPVbyte_or_null"

Like "SvPVbyte", but when "sv" is undef, returns "NULL".

```
 char*  SvPVbyte_or_null(SV* sv, STRLEN len)
```

"SvPVbyte_or_null_nomg"

Like "SvPVbyte_or_null", but does not process get magic.

```
 char*  SvPVbyte_or_null_nomg(SV* sv, STRLEN len)
```

"SvPVCLEAR"

Ensures that sv is a SVt_PV and that its SvCUR is 0, and that it is properly null terminated. Equivalent to sv_setpvs(""), but more efficient.

```
 char *  SvPVCLEAR(SV* sv)
```

"SvPV_force"

"SvPV_force_nolen"

"SvPVx_force"

"SvPV_force_nomg"

"SvPV_force_nomg_nolen"

"SvPV_force_mutable"

"SvPV_force_flags"

"SvPV_force_flags_nolen"

"SvPV_force_flags_mutable"

"SvPVbyte_force"

"SvPVbytex_force"

"SvPVutf8_force"

"SvPVutf8x_force"

These are like "SvPV", returning the string in the SV, but will force the SV into containing a string ("SvPOK"), and only a string ("SvPOK_only"), by hook or by crook. You need to use one of these "force" routines if you are going to update the "SvPVX" directly.

Note that coercing an arbitrary scalar into a plain PV will potentially strip useful data from it. For example if the SV was "SvROK", then the referent will have its reference count decremented, and the SV itself may be converted to an "SvPOK" scalar with a string buffer containing a value such as "ARRAY(0x1234)".

The differences between the forms are:

The forms with "flags" in their names allow you to use the "flags" parameter to specify to perform 'get' magic (by setting the "SV_GMAGIC" flag) or to skip 'get' magic (by clearing it). The other forms do perform 'get' magic, except for the ones with "nomg" in their names, which skip 'get' magic.

The forms that take a "len" parameter will set that variable to the byte length of the resultant string (these are macros, so don't use &len).

The forms with "nolen" in their names indicate they don't have a "len" parameter. They should be used only when it is known that the PV is a C string, terminated by a NUL byte, and without intermediate NUL characters; or when you don't care about its length.

The forms with "mutable" in their names are effectively the same as those without, but the name emphasizes that the string is modifiable by the caller, which it is in all the forms.

"SvPVutf8_force" is like "SvPV_force", but converts "sv" to UTF-8 first if not already UTF-8.

"SvPVutf8x_force" is like "SvPVutf8_force", but guarantees to evaluate "sv" only once; use the more efficient "SvPVutf8_force" otherwise.

"SvPVbyte_force" is like "SvPV_force", but converts "sv" to byte representation first if currently encoded as UTF-8. If the SV cannot be downgraded from UTF-8, this croaks.

"SvPVbytex_force" is like "SvPVbyte_force", but guarantees to evaluate "sv" only once; use the more efficient "SvPVbyte_force" otherwise.

```
 char*  SvPV_force              (SV* sv, STRLEN len)
 char*  SvPV_force_nolen        (SV* sv)
 char*  SvPVx_force             (SV* sv, STRLEN len)
 char*  SvPV_force_nomg         (SV* sv, STRLEN len)
 char*  SvPV_force_nomg_nolen   (SV * sv)
 char*  SvPV_force_mutable      (SV * sv, STRLEN len)
```

char* SvPV_force_flags (SV * sv, STRLEN len, U32 flags)

char* SvPV_force_flags_nolen (SV * sv, U32 flags)

char* SvPV_force_flags_mutable(SV * sv, STRLEN len, U32 flags)

char* SvPVbyte_force (SV* sv, STRLEN len)

char* SvPVbytex_force (SV* sv, STRLEN len)

char* SvPVutf8_force (SV* sv, STRLEN len)

char* SvPVutf8x_force (SV* sv, STRLEN len)

"SvPV_free"

Frees the PV buffer in "sv", leaving things in a precarious state, so should only be used as part of a larger operation

void SvPV_free(SV * sv)

"sv_pvn_force_flags"

Get a sensible string out of the SV somehow. If "flags" has the "SV_GMAGIC" bit set, will "mg_get" on "sv" if appropriate, else not. "sv_pvn_force" and "sv_pvn_force_nomg" are implemented in terms of this function. You normally want to use the various wrapper macros instead: see "SvPV_force" and "SvPV_force_nomg".

char* sv_pvn_force_flags(SV *const sv, STRLEN *const lp,
const U32 flags)

"SvPV_renew"

Low level micro optimization of "SvGROW". It is generally better to use "SvGROW" instead. This is because "SvPV_renew" ignores potential issues that "SvGROW" handles. "sv" needs to have a real "PV" that is unencombered by things like COW. Using "SV_CHECK_THINKFIRST" or "SV_CHECK_THINKFIRST_COW_DROP" before calling this should clean it up, but why not just use "SvGROW" if you're not sure about the provenance?

void SvPV_renew(SV* sv, STRLEN len)

"SvPV_set"

This is probably not what you want to use, you probably wanted "sv_usepvn_flags" or "sv_setpvn" or "sv_setpvs".

Set the value of the PV pointer in "sv" to the Perl allocated "NUL"-terminated string "val". See also "SvIV_set".

Remember to free the previous PV buffer. There are many things to check. Beware that the existing pointer may be involved in copy-on-write or other mischief, so do "SvOOK_off(sv)" and use "sv_force_normal" or "SvPV_force" (or check the "SvIsCOW"

flag) first to make sure this modification is safe. Then finally, if it is not a COW, call "SvPV_free" to free the previous PV buffer.

```
 void  SvPV_set(SV* sv, char* val)
```

"SvPVutf8"

Like "SvPV", but converts "sv" to UTF-8 first if necessary.

```
 char*  SvPVutf8(SV* sv, STRLEN len)
```

"SvPVutf8_force"

Like "SvPV_force", but converts "sv" to UTF-8 first if necessary.

```
 char*  SvPVutf8_force(SV* sv, STRLEN len)
```

"SvPVutf8_nolen"

Like "SvPV_nolen", but converts "sv" to UTF-8 first if necessary.

```
 char*  SvPVutf8_nolen(SV* sv)
```

"SvPVutf8_nomg"

Like "SvPVutf8", but does not process get magic.

```
 char*  SvPVutf8_nomg(SV* sv, STRLEN len)
```

"SvPVutf8_or_null"

Like "SvPVutf8", but when "sv" is undef, returns "NULL".

```
 char*  SvPVutf8_or_null(SV* sv, STRLEN len)
```

"SvPVutf8_or_null_nomg"

Like "SvPVutf8_or_null", but does not process get magic.

```
 char*  SvPVutf8_or_null_nomg(SV* sv, STRLEN len)
```

"SvPVX"

"SvPVXx"

"SvPVX_const"

"SvPVX_mutable"

These return a pointer to the physical string in the SV. The SV must contain a string. Prior to 5.9.3 it is not safe to execute these unless the SV's type >= "SVt_PV".

These are also used to store the name of an autoloaded subroutine in an XS AUTOLOAD routine. See "Autoloading with XSUBs" in perlguts.

"SvPVXx" is identical to "SvPVX".

"SvPVX_mutable" is merely a synonym for "SvPVX", but its name emphasizes that the string is modifiable by the caller.

"SvPVX_const" differs in that the return value has been cast so that the compiler will complain if you were to try to modify the contents of the string, (unless you cast away const yourself).

```
char*       SvPVX       (SV* sv)
char*       SvPVXx      (SV* sv)
const char* SvPVX_const (SV* sv)
char*       SvPVX_mutable(SV* sv)
```

"SvREADONLY"

Returns true if the argument is readonly, otherwise returns false. Exposed to perl code via Internals::SvREADONLY().

```
U32 SvREADONLY(SV* sv)
```

"SvREADONLY_off"

Mark an object as not-readonly. Exactly what this mean depends on the object type. Exposed to perl code via Internals::SvREADONLY().

```
U32 SvREADONLY_off(SV* sv)
```

"SvREADONLY_on"

Mark an object as readonly. Exactly what this means depends on the object type. Exposed to perl code via Internals::SvREADONLY().

```
U32 SvREADONLY_on(SV* sv)
```

"sv_ref"

Returns a SV describing what the SV passed in is a reference to. dst can be a SV to be set to the description or NULL, in which case a mortal SV is returned.

If ob is true and the SV is blessed, the description is the class name, otherwise it is the type of the SV, "SCALAR", "ARRAY" etc.

```
SV* sv_ref(SV *dst, const SV *const sv, const int ob)
```

"SvREFCNT"

Returns the value of the object's reference count. Exposed to perl code via Internals::SvREFCNT().

```
U32 SvREFCNT(SV* sv)
```

"SvREFCNT_dec"

"SvREFCNT_dec_NN"

These decrement the reference count of the given SV.

"SvREFCNT_dec_NN" may only be used when "sv" is known to not be "NULL".

```
void  SvREFCNT_dec(SV *sv)
```

"SvREFCNT_inc"

"SvREFCNT_inc_NN"

"SvREFCNT_inc_void"

"SvREFCNT_inc_void_NN"

"SvREFCNT_inc_simple"

"SvREFCNT_inc_simple_NN"

"SvREFCNT_inc_simple_void"

"SvREFCNT_inc_simple_void_NN"

These all increment the reference count of the given SV. The ones without "void" in their names return the SV.

"SvREFCNT_inc" is the base operation; the rest are optimizations if various input constraints are known to be true; hence, all can be replaced with "SvREFCNT_inc".

"SvREFCNT_inc_NN" can only be used if you know "sv" is not "NULL". Since we don't have to check the NULLness, it's faster and smaller.

"SvREFCNT_inc_void" can only be used if you don't need the return value. The macro doesn't need to return a meaningful value.

"SvREFCNT_inc_void_NN" can only be used if you both don't need the return value, and you know that "sv" is not "NULL". The macro doesn't need to return a meaningful value, or check for NULLness, so it's smaller and faster.

"SvREFCNT_inc_simple" can only be used with expressions without side effects. Since we don't have to store a temporary value, it's faster.

"SvREFCNT_inc_simple_NN" can only be used with expressions without side effects and you know "sv" is not "NULL". Since we don't have to store a temporary value, nor check for NULLness, it's faster and smaller.

"SvREFCNT_inc_simple_void" can only be used with expressions without side effects and you don't need the return value.

"SvREFCNT_inc_simple_void_NN" can only be used with expressions without side effects, you don't need the return value, and you know "sv" is not "NULL".

```
SV *  SvREFCNT_inc            (SV *sv)
SV *  SvREFCNT_inc_NN         (SV *sv)
void  SvREFCNT_inc_void       (SV *sv)
```

```
 void  SvREFCNT_inc_void_NN       (SV* sv)
 SV*   SvREFCNT_inc_simple        (SV* sv)
 SV*   SvREFCNT_inc_simple_NN     (SV* sv)
 void  SvREFCNT_inc_simple_void   (SV* sv)
 void  SvREFCNT_inc_simple_void_NN(SV* sv)
```

"sv_reftype"

Returns a string describing what the SV is a reference to.

If ob is true and the SV is blessed, the string is the class name, otherwise it is the type of the SV, "SCALAR", "ARRAY" etc.

```
 const char*  sv_reftype(const SV *const sv, const int ob)
```

"sv_replace"

Make the first argument a copy of the second, then delete the original. The target SV physically takes over ownership of the body of the source SV and inherits its flags; however, the target keeps any magic it owns, and any magic in the source is discarded. Note that this is a rather specialist SV copying operation; most of the time you'll want to use "sv_setsv" or one of its many macro front-ends.

```
 void  sv_replace(SV *const sv, SV *const nsv)
```

"sv_report_used"

Dump the contents of all SVs not yet freed (debugging aid).

```
 void  sv_report_used()
```

"sv_reset"

Underlying implementation for the "reset" Perl function. Note that the perl-level function is vaguely deprecated.

```
 void  sv_reset(const char* s, HV *const stash)
```

"SvROK"

Tests if the SV is an RV.

```
 U32  SvROK(SV* sv)
```

"SvROK_off"

Unsets the RV status of an SV.

```
 void  SvROK_off(SV* sv)
```

"SvROK_on"

Tells an SV that it is an RV.

```
 void  SvROK_on(SV* sv)
```

"SvRV"

Dereferences an RV to return the SV.

```
SV*  SvRV(SV* sv)
```

"SvRV_set"

Set the value of the RV pointer in "sv" to val. See "SvIV_set".

```
void  SvRV_set(SV* sv, SV* val)
```

"sv_rvunweaken"

Unweaken a reference: Clear the "SvWEAKREF" flag on this RV; remove the backreference to this RV from the array of backreferences associated with the target SV, increment the refcount of the target. Silently ignores "undef" and warns on non-weak references.

```
SV*  sv_rvunweaken(SV *const sv)
```

"sv_rvweaken"

Weaken a reference: set the "SvWEAKREF" flag on this RV; give the referred-to SV "PERL_MAGIC_backref" magic if it hasn't already; and push a back-reference to this RV onto the array of backreferences associated with that magic. If the RV is magical, set magic will be called after the RV is cleared. Silently ignores "undef" and warns on already-weak references.

```
SV*  sv_rvweaken(SV *const sv)
```

"sv_setiv"

"sv_setiv_mg"

These copy an integer into the given SV, upgrading first if necessary.

They differ only in that "sv_setiv_mg" handles 'set' magic; "sv_setiv" does not.

```
void  sv_setiv   (SV *const sv, const IV num)
void  sv_setiv_mg(SV *const sv, const IV i)
```

"SvSETMAGIC"

Invokes "mg_set" on an SV if it has 'set' magic. This is necessary after modifying a scalar, in case it is a magical variable like $| or a tied variable (it calls "STORE"). This macro evaluates its argument more than once.

```
void  SvSETMAGIC(SV* sv)
```

"sv_setnv"

"sv_setnv_mg"

These copy a double into the given SV, upgrading first if necessary.

They differ only in that "sv_setnv_mg" handles 'set' magic; "sv_setnv" does not.

```
 void  sv_setnv(SV *const sv, const NV num)
```

"sv_setpv"

"sv_setpv_mg"

These copy a string into an SV. The string must be terminated with a "NUL" character, and not contain embeded "NUL"'s.

They differ only in that:

"sv_setpv" does not handle 'set' magic; "sv_setpv_mg" does.

```
 void  sv_setpv(SV *const sv, const char *const ptr)
```

"sv_setpvf"

"sv_setpvf_nocontext"

"sv_setpvf_mg"

"sv_setpvf_mg_nocontext"

These work like "sv_catpvf" but copy the text into the SV instead of appending it.

The differences between these are:

"sv_setpvf" and "sv_setpvf_nocontext" do not handle 'set' magic; "sv_setpvf_mg" and "sv_setpvf_mg_nocontext" do.

"sv_setpvf_nocontext" and "sv_setpvf_mg_nocontext" do not take a thread context ("aTHX") parameter, so are used in situations where the caller doesn't already have the thread context.

NOTE: "sv_setpvf" must be explicitly called as "Perl_sv_setpvf" with an "aTHX_" parameter.

NOTE: "sv_setpvf_mg" must be explicitly called as "Perl_sv_setpvf_mg" with an "aTHX_" parameter.

```
 void  Perl_sv_setpvf       (pTHX_ SV *const sv,
                             const char *const pat, ...)
 void  sv_setpvf_nocontext   (SV *const sv, const char *const pat,
                             ...)
 void  Perl_sv_setpvf_mg     (pTHX_ SV *const sv,
                             const char *const pat, ...)
 void  sv_setpvf_mg_nocontext(SV *const sv, const char *const pat,
                             ...)
```

"sv_setpviv"

"sv_setpviv_mg"

"DEPRECATED!" It is planned to remove "sv_setpviv" from a future release of Perl. Do not use it for new code; remove it from existing code.

"DEPRECATED!" It is planned to remove "sv_setpviv_mg" from a future release of Perl. Do not use it for new code; remove it from existing code.

These copy an integer into the given SV, also updating its string value.

They differ only in that "sv_setpviv_mg" performs 'set' magic; "sv_setpviv" skips any magic.

```
void  sv_setpviv   (SV *const sv, const IV num)
void  sv_setpviv_mg(SV *const sv, const IV iv)
```

"sv_setpvn"

"sv_setpvn_mg"

These copy a string (possibly containing embedded "NUL" characters) into an SV. The "len" parameter indicates the number of bytes to be copied. If the "ptr" argument is NULL the SV will become undefined.

The UTF-8 flag is not changed by these functions. A terminating NUL byte is guaranteed.

They differ only in that:

"sv_setpvn" does not handle 'set' magic; "sv_setpvn_mg" does.

```
void  sv_setpvn(SV *const sv, const char *const ptr,
                const STRLEN len)
```

"sv_setpvs"

Like "sv_setpvn", but takes a literal string instead of a string/length pair.

```
void  sv_setpvs(SV* sv, "literal string")
```

"sv_setpvs_mg"

Like "sv_setpvn_mg", but takes a literal string instead of a string/length pair.

```
void  sv_setpvs_mg(SV* sv, "literal string")
```

"sv_setpv_bufsize"

Sets the SV to be a string of cur bytes length, with at least len bytes available.

Ensures that there is a null byte at SvEND. Returns a char * pointer to the SvPV buffer.

```
char  *  sv_setpv_bufsize(SV *const sv, const STRLEN cur,
                          const STRLEN len)
```

"sv_setref_iv"

Copies an integer into a new SV, optionally blessing the SV. The "rv" argument will be upgraded to an RV. That RV will be modified to point to the new SV. The "classname" argument indicates the package for the blessing. Set "classname" to "NULL" to avoid the blessing. The new SV will have a reference count of 1, and the RV will be returned.

```
SV*  sv_setref_iv(SV *const rv, const char *const classname,
                  const IV iv)
```

"sv_setref_nv"

Copies a double into a new SV, optionally blessing the SV. The "rv" argument will be upgraded to an RV. That RV will be modified to point to the new SV. The "classname" argument indicates the package for the blessing. Set "classname" to "NULL" to avoid the blessing. The new SV will have a reference count of 1, and the RV will be returned.

```
SV*  sv_setref_nv(SV *const rv, const char *const classname,
                  const NV nv)
```

"sv_setref_pv"

Copies a pointer into a new SV, optionally blessing the SV. The "rv" argument will be upgraded to an RV. That RV will be modified to point to the new SV. If the "pv" argument is "NULL", then "PL_sv_undef" will be placed into the SV. The "classname" argument indicates the package for the blessing. Set "classname" to "NULL" to avoid the blessing. The new SV will have a reference count of 1, and the RV will be returned.

Do not use with other Perl types such as HV, AV, SV, CV, because those objects will become corrupted by the pointer copy process.

Note that "sv_setref_pvn" copies the string while this copies the pointer.

```
SV*  sv_setref_pv(SV *const rv, const char *const classname,
                  void *const pv)
```

"sv_setref_pvn"

Copies a string into a new SV, optionally blessing the SV. The length of the string must be specified with "n". The "rv" argument will be upgraded to an RV. That RV will be modified to point to the new SV. The "classname" argument indicates the package for the blessing. Set "classname" to "NULL" to avoid the blessing. The new

SV will have a reference count of 1, and the RV will be returned.

Note that "sv_setref_pv" copies the pointer while this copies the string.

```
 SV*  sv_setref_pvn(SV *const rv, const char *const classname,
                    const char *const pv, const STRLEN n)
```

"sv_setref_pvs"

Like "sv_setref_pvn", but takes a literal string instead of a string/length pair.

```
 SV *  sv_setref_pvs(SV *const rv, const char *const classname,
                     "literal string")
```

"sv_setref_uv"

Copies an unsigned integer into a new SV, optionally blessing the SV. The "rv" argument will be upgraded to an RV. That RV will be modified to point to the new SV. The "classname" argument indicates the package for the blessing. Set "classname" to "NULL" to avoid the blessing. The new SV will have a reference count of 1, and the RV will be returned.

```
 SV*  sv_setref_uv(SV *const rv, const char *const classname,
                   const UV uv)
```

"SvSetSV"

"SvSetMagicSV"

"SvSetSV_nosteal"

"SvSetMagicSV_nosteal"

if "dsv" is the same as "ssv", these do nothing. Otherwise they all call some form of "sv_setsv". They may evaluate their arguments more than once.

The only differences are:

"SvSetMagicSV" and "SvSetMagicSV_nosteal" perform any required 'set' magic afterwards on the destination SV; "SvSetSV" and "SvSetSV_nosteal" do not.

"SvSetSV_nosteal" "SvSetMagicSV_nosteal" call a non-destructive version of "sv_setsv".

```
 void  SvSetSV(SV* dsv, SV* ssv)
```

"sv_setsv"

"sv_setsv_flags"

"sv_setsv_mg"

"sv_setsv_nomg"

These copy the contents of the source SV "ssv" into the destination SV "dsv". "ssv" may be destroyed if it is mortal, so don't use these functions if the source SV needs

to be reused. Loosely speaking, they perform a copy-by-value, obliterating any previous content of the destination.

They differ only in that:

"sv_setsv" calls 'get' magic on "ssv", but skips 'set' magic on "dsv".

"sv_setsv_mg" calls both 'get' magic on "ssv" and 'set' magic on "dsv".

"sv_setsv_nomg" skips all magic.

"sv_setsv_flags" has a "flags" parameter which you can use to specify any combination of magic handling, and also you can specify "SV_NOSTEAL" so that the buffers of temps will not be stolen.

You probably want to instead use one of the assortment of wrappers, such as "SvSetSV", "SvSetSV_nosteal", "SvSetMagicSV" and "SvSetMagicSV_nosteal".

"sv_setsv_flags" is the primary function for copying scalars, and most other copy-ish functions and macros use it underneath.

```
 void  sv_setsv      (SV *dsv, SV *ssv)
 void  sv_setsv_flags(SV *dsv, SV *ssv, const I32 flags)
 void  sv_setsv_mg   (SV *const dsv, SV *const ssv)
 void  sv_setsv_nomg (SV *dsv, SV *ssv)
```

"sv_setuv"

"sv_setuv_mg"

These copy an unsigned integer into the given SV, upgrading first if necessary.

They differ only in that "sv_setuv_mg" handles 'set' magic; "sv_setuv" does not.

```
 void  sv_setuv   (SV *const sv, const UV num)
 void  sv_setuv_mg(SV *const sv, const UV u)
```

"sv_set_undef"

Equivalent to "sv_setsv(sv, &PL_sv_undef)", but more efficient. Doesn't handle set magic.

The perl equivalent is "$sv = undef;". Note that it doesn't free any string buffer, unlike "undef $sv".

Introduced in perl 5.25.12.

```
 void  sv_set_undef(SV *sv)
```

"SvSHARE"

Arranges for "sv" to be shared between threads if a suitable module has been loaded.

```
 void  SvSHARE(SV* sv)
```

"SvSHARED_HASH"

Returns the hash for "sv" created by "newSVpvn_share".

```
struct hek*  SvSHARED_HASH(SV * sv)
```

"SvSTASH"

Returns the stash of the SV.

```
HV*  SvSTASH(SV* sv)
```

"SvSTASH_set"

Set the value of the STASH pointer in "sv" to val. See "SvIV_set".

```
void  SvSTASH_set(SV* sv, HV* val)
```

"SvTAINT"

Taints an SV if tainting is enabled, and if some input to the current expression is tainted--usually a variable, but possibly also implicit inputs such as locale settings. "SvTAINT" propagates that taintedness to the outputs of an expression in a pessimistic fashion; i.e., without paying attention to precisely which outputs are influenced by which inputs.

```
void  SvTAINT(SV* sv)
```

"SvTAINTED"

Checks to see if an SV is tainted. Returns TRUE if it is, FALSE if not.

```
bool  SvTAINTED(SV* sv)
```

"SvTAINTED_off"

Untaints an SV. Be very careful with this routine, as it short-circuits some of Perl's fundamental security features. XS module authors should not use this function unless they fully understand all the implications of unconditionally untainting the value. Untainting should be done in the standard perl fashion, via a carefully crafted regexp, rather than directly untainting variables.

```
void  SvTAINTED_off(SV* sv)
```

"SvTAINTED_on"

Marks an SV as tainted if tainting is enabled.

```
void  SvTAINTED_on(SV* sv)
```

"SvTRUE"

"SvTRUEx"

"SvTRUE_nomg"

"SvTRUE_NN"

"SvTRUE_nomg_NN"

These return a boolean indicating whether Perl would evaluate the SV as true or false.

See "SvOK" for a defined/undefined test.

As of Perl 5.32, all are guaranteed to evaluate "sv" only once. Prior to that release, only "SvTRUEx" guaranteed single evaluation; now "SvTRUEx" is identical to "SvTRUE".

"SvTRUE_nomg" and "TRUE_nomg_NN" do not perform 'get' magic; the others do unless the scalar is already "SvPOK", "SvIOK", or "SvNOK" (the public, not the private flags).

"SvTRUE_NN" is like "SvTRUE", but "sv" is assumed to be non-null (NN). If there is a possibility that it is NULL, use plain "SvTRUE".

"SvTRUE_nomg_NN" is like "SvTRUE_nomg", but "sv" is assumed to be non-null (NN). If there is a possibility that it is NULL, use plain "SvTRUE_nomg".

```
bool  SvTRUE(SV *sv)
```

"SvTYPE"

Returns the type of the SV. See "svtype".

```
svtype  SvTYPE(SV* sv)
```

"SvUNLOCK"

Releases a mutual exclusion lock on "sv" if a suitable module has been loaded.

```
void  SvUNLOCK(SV* sv)
```

"sv_unmagic"

Removes all magic of type "type" from an SV.

```
int  sv_unmagic(SV *const sv, const int type)
```

"sv_unmagicext"

Removes all magic of type "type" with the specified "vtbl" from an SV.

```
int  sv_unmagicext(SV *const sv, const int type, MGVTBL *vtbl)
```

"sv_unref"

Unsets the RV status of the SV, and decrements the reference count of whatever was being referenced by the RV. This can almost be thought of as a reversal of "newSVrv".

This is "sv_unref_flags" with the "flag" being zero. See "SvROK_off".

```
void  sv_unref(SV* sv)
```

"sv_unref_flags"

Unsets the RV status of the SV, and decrements the reference count of whatever was being referenced by the RV. This can almost be thought of as a reversal of "newSVrv".

The "cflags" argument can contain "SV_IMMEDIATE_UNREF" to force the reference count to be decremented (otherwise the decrementing is conditional on the reference count being different from one or the reference being a readonly SV). See "SvROK_off".

```
void  sv_unref_flags(SV *const ref, const U32 flags)
```

"SvUOK"

Returns a boolean indicating whether the SV contains an integer that must be interpreted as unsigned. A non-negative integer whose value is within the range of both an IV and a UV may be flagged as either "SvUOK" or "SvIOK".

```
bool  SvUOK(SV* sv)
```

"SvUPGRADE"

Used to upgrade an SV to a more complex form. Uses "sv_upgrade" to perform the upgrade if necessary. See "svtype".

```
void  SvUPGRADE(SV* sv, svtype type)
```

"sv_upgrade"

Upgrade an SV to a more complex form. Generally adds a new body type to the SV, then copies across as much information as possible from the old body. It croaks if the SV is already in a more complex form than requested. You generally want to use the "SvUPGRADE" macro wrapper, which checks the type before calling "sv_upgrade", and hence does not croak. See also "svtype".

```
void  sv_upgrade(SV *const sv, svtype new_type)
```

"sv_usepvn"

Tells an SV to use "ptr" to find its string value. Implemented by calling "sv_usepvn_flags" with "flags" of 0, hence does not handle 'set' magic. See "sv_usepvn_flags".

```
void  sv_usepvn(SV* sv, char* ptr, STRLEN len)
```

"sv_usepvn_flags"

Tells an SV to use "ptr" to find its string value. Normally the string is stored inside the SV, but sv_usepvn allows the SV to use an outside string. "ptr" should point to memory that was allocated by "Newx". It must be the start of a "Newx"-ed block of memory, and not a pointer to the middle of it (beware of "OOK" and copy-on-write), and not be from a non-"Newx" memory allocator like "malloc". The string length, "len", must be supplied. By default this function will "Renew" (i.e. realloc, move) the memory pointed to by "ptr", so that pointer should not be freed or used by

the programmer after giving it to "sv_usepvn", and neither should any pointers from "behind" that pointer (e.g. ptr + 1) be used.

If "flags?&?SV_SMAGIC" is true, will call "SvSETMAGIC". If "flags?&?SV_HAS_TRAILING_NUL" is true, then "ptr[len]" must be "NUL", and the realloc will be skipped (i.e. the buffer is actually at least 1 byte longer than "len", and already meets the requirements for storing in "SvPVX").

```
 void  sv_usepvn_flags(SV *const sv, char* ptr, const STRLEN len,
                       const U32 flags)
```

"sv_usepvn_mg"

Like "sv_usepvn", but also handles 'set' magic.

```
 void  sv_usepvn_mg(SV *sv, char *ptr, STRLEN len)
```

"SvUTF8"

Returns a U32 value indicating the UTF-8 status of an SV. If things are set-up properly, this indicates whether or not the SV contains UTF-8 encoded data. You should use this after a call to "SvPV" or one of its variants, in case any call to string overloading updates the internal flag.

If you want to take into account the bytes pragma, use "DO_UTF8" instead.

```
 U32  SvUTF8(SV* sv)
```

"sv_utf8_decode"

If the PV of the SV is an octet sequence in Perl's extended UTF-8 and contains a multiple-byte character, the "SvUTF8" flag is turned on so that it looks like a character. If the PV contains only single-byte characters, the "SvUTF8" flag stays off. Scans PV for validity and returns FALSE if the PV is invalid UTF-8.

```
 bool  sv_utf8_decode(SV *const sv)
```

"sv_utf8_downgrade"

"sv_utf8_downgrade_flags"

"sv_utf8_downgrade_nomg"

These attempt to convert the PV of an SV from characters to bytes. If the PV contains a character that cannot fit in a byte, this conversion will fail; in this case, "FALSE" is returned if "fail_ok" is true; otherwise they croak.

They are not a general purpose Unicode to byte encoding interface: use the "Encode" extension for that.

They differ only in that:

"sv_utf8_downgrade" processes 'get' magic on "sv".

"sv_utf8_downgrade_nomg" does not.

"sv_utf8_downgrade_flags" has an additional "flags" parameter in which you can specify "SV_GMAGIC" to process 'get' magic, or leave it cleared to not proccess 'get' magic.

```
 bool  sv_utf8_downgrade      (SV *const sv, const bool fail_ok)
 bool  sv_utf8_downgrade_flags(SV *const sv, const bool fail_ok,
                               const U32 flags)
 bool  sv_utf8_downgrade_nomg (SV *const sv, const bool fail_ok)
```

"sv_utf8_encode"

Converts the PV of an SV to UTF-8, but then turns the "SvUTF8" flag off so that it looks like octets again.

```
 void  sv_utf8_encode(SV *const sv)
```

"sv_utf8_upgrade"

"sv_utf8_upgrade_nomg"

"sv_utf8_upgrade_flags"

"sv_utf8_upgrade_flags_grow"

These convert the PV of an SV to its UTF-8-encoded form. The SV is forced to string form if it is not already. They always set the "SvUTF8" flag to avoid future validity checks even if the whole string is the same in UTF-8 as not. They return the number of bytes in the converted string

The forms differ in just two ways. The main difference is whether or not they perform 'get magic' on "sv". "sv_utf8_upgrade_nomg" skips 'get magic'; "sv_utf8_upgrade" performs it; and "sv_utf8_upgrade_flags" and "sv_utf8_upgrade_flags_grow" either perform it (if the "SV_GMAGIC" bit is set in "flags") or don't (if that bit is cleared).

The other difference is that "sv_utf8_upgrade_flags_grow" has an additional parameter, "extra", which allows the caller to specify an amount of space to be reserved as spare beyond what is needed for the actual conversion. This is used when the caller knows it will soon be needing yet more space, and it is more efficient to request space from the system in a single call. This form is otherwise identical to "sv_utf8_upgrade_flags".

These are not a general purpose byte encoding to Unicode interface: use the Encode extension for that.

The "SV_FORCE_UTF8_UPGRADE" flag is now ignored.

```
STRLEN  sv_utf8_upgrade          (SV *sv)
STRLEN  sv_utf8_upgrade_nomg     (SV *sv)
STRLEN  sv_utf8_upgrade_flags    (SV *const sv, const I32 flags)
STRLEN  sv_utf8_upgrade_flags_grow(SV *const sv, const I32 flags,
                                   STRLEN extra)
```

"SvUTF8_off"

Unsets the UTF-8 status of an SV (the data is not changed, just the flag). Do not use frivolously.

```
void  SvUTF8_off(SV *sv)
```

"SvUTF8_on"

Turn on the UTF-8 status of an SV (the data is not changed, just the flag). Do not use frivolously.

```
void  SvUTF8_on(SV *sv)
```

"SvUV"

"SvUVx"

"SvUV_nomg"

These coerce the given SV to UV and return it. The returned value in many circumstances will get stored in "sv"'s UV slot, but not in all cases. (Use "sv_setuv" to make sure it does).

"SvUVx" is different from the others in that it is guaranteed to evaluate "sv" exactly once; the others may evaluate it multiple times. Only use this form if "sv" is an expression with side effects, otherwise use the more efficient "SvUV".

"SvUV_nomg" is the same as "SvUV", but does not perform 'get' magic.

```
UV  SvUV(SV* sv)
```

"SvUV_set"

Set the value of the UV pointer in "sv" to val. See "SvIV_set".

```
void  SvUV_set(SV* sv, UV val)
```

"SvUVX"

Returns the raw value in the SV's UV slot, without checks or conversions. Only use when you are sure "SvIOK" is true. See also "SvUV".

```
UV  SvUVX(SV* sv)
```

"SvUVXx"

"DEPRECATED!" It is planned to remove "SvUVXx" from a future release of Perl. Do not use it for new code; remove it from existing code.

This is an unnecessary synonym for "SvUVX"

```
UV  SvUVXx(SV* sv)
```

"sv_vcatpvf"

"sv_vcatpvf_mg"

These process their arguments like "sv_vcatpvfn" called with a non-null C-style variable argument list, and append the formatted output to "sv".

They differ only in that "sv_vcatpvf_mg" performs 'set' magic; "sv_vcatpvf" skips 'set' magic.

Both perform 'get' magic.

They are usually accessed via their frontends "sv_catpvf" and "sv_catpvf_mg".

```
void  sv_vcatpvf(SV *const sv, const char *const pat,
                 va_list *const args)
```

"sv_vcatpvfn"

"sv_vcatpvfn_flags"

These process their arguments like vsprintf(3) and append the formatted output to an SV. They use an array of SVs if the C-style variable argument list is missing ("NULL"). Argument reordering (using format specifiers like "%2$d" or "%*2$d") is supported only when using an array of SVs; using a C-style "va_list" argument list with a format string that uses argument reordering will yield an exception.

When running with taint checks enabled, they indicate via "maybe_tainted" if results are untrustworthy (often due to the use of locales).

They assume that "pat" has the same utf8-ness as "sv". It's the caller's responsibility to ensure that this is so.

They differ in that "sv_vcatpvfn_flags" has a "flags" parameter in which you can set or clear the "SV_GMAGIC" and/or SV_SMAGIC flags, to specify which magic to handle or not handle; whereas plain "sv_vcatpvfn" always specifies both 'get' and 'set' magic.

They are usually used via one of the frontends "sv_vcatpvf" and "sv_vcatpvf_mg".

```
void  sv_vcatpvfn       (SV *const sv, const char *const pat,
                         const STRLEN patlen, va_list *const args,
                         SV **const svargs, const Size_t sv_count,
                         bool *const maybe_tainted)
```

```
 void  sv_vcatpvfn_flags(SV *const sv, const char *const pat,
                         const STRLEN patlen, va_list *const args,
                         SV **const svargs, const Size_t sv_count,
                         bool *const maybe_tainted,
                         const U32 flags)
```

"SvVOK"

Returns a boolean indicating whether the SV contains a v-string.

```
 bool  SvVOK(SV* sv)
```

"sv_vsetpvf"

"sv_vsetpvf_mg"

These work like "sv_vcatpvf" but copy the text into the SV instead of appending it.

They differ only in that "sv_vsetpvf_mg" performs 'set' magic; "sv_vsetpvf" skips all magic.

They are usually used via their frontends, "sv_setpvf" and "sv_setpvf_mg".

```
 void  sv_vsetpvf(SV *const sv, const char *const pat,
                  va_list *const args)
```

"sv_vsetpvfn"

Works like "sv_vcatpvfn" but copies the text into the SV instead of appending it.

Usually used via one of its frontends "sv_vsetpvf" and "sv_vsetpvf_mg".

```
 void  sv_vsetpvfn(SV *const sv, const char *const pat,
                   const STRLEN patlen, va_list *const args,
                   SV **const svargs, const Size_t sv_count,
                   bool *const maybe_tainted)
```

"SvVSTRING_mg"

Returns the vstring magic, or NULL if none

```
 MAGIC*  SvVSTRING_mg(SV * sv)
```

"vnewSVpvf"

Like "newSVpvf" but but the arguments are an encapsulated argument list.

```
 SV*  vnewSVpvf(const char *const pat, va_list *const args)
```

Time

"ASCTIME_R_PROTO"

This symbol encodes the prototype of "asctime_r". It is zero if "d_asctime_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_asctime_r" is

defined.

"CTIME_R_PROTO"

This symbol encodes the prototype of "ctime_r". It is zero if "d_ctime_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_ctime_r" is defined.

"GMTIME_MAX"

This symbol contains the maximum value for the "time_t" offset that the system function gmtime () accepts, and defaults to 0

"GMTIME_MIN"

This symbol contains the minimum value for the "time_t" offset that the system function gmtime () accepts, and defaults to 0

"GMTIME_R_PROTO"

This symbol encodes the prototype of "gmtime_r". It is zero if "d_gmtime_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_gmtime_r" is defined.

"HAS_ASCTIME64"

This symbol, if defined, indicates that the "asctime64" () routine is available to do the 64bit variant of asctime ()

"HAS_ASCTIME_R"

This symbol, if defined, indicates that the "asctime_r" routine is available to asctime re-entrantly.

"HAS_CTIME64"

This symbol, if defined, indicates that the "ctime64" () routine is available to do the 64bit variant of ctime ()

"HAS_CTIME_R"

This symbol, if defined, indicates that the "ctime_r" routine is available to ctime re-entrantly.

"HAS_DIFFTIME"

This symbol, if defined, indicates that the "difftime" routine is available.

"HAS_DIFFTIME64"

This symbol, if defined, indicates that the "difftime64" () routine is available to do the 64bit variant of difftime ()

"HAS_FUTIMES"

This symbol, if defined, indicates that the "futimes" routine is available to change file descriptor time stamps with "struct timevals".

"HAS_GETITIMER"

This symbol, if defined, indicates that the "getitimer" routine is available to return interval timers.

"HAS_GETTIMEOFDAY"

This symbol, if defined, indicates that the "gettimeofday()" system call is available for a sub-second accuracy clock. Usually, the file sys/resource.h needs to be included (see "I_SYS_RESOURCE"). The type "Timeval" should be used to refer to ""struct timeval"".

"HAS_GMTIME64"

This symbol, if defined, indicates that the "gmtime64" () routine is available to do the 64bit variant of gmtime ()

"HAS_GMTIME_R"

This symbol, if defined, indicates that the "gmtime_r" routine is available to gmtime re-entrantly.

"HAS_LOCALTIME64"

This symbol, if defined, indicates that the "localtime64" () routine is available to do the 64bit variant of localtime ()

"HAS_LOCALTIME_R"

This symbol, if defined, indicates that the "localtime_r" routine is available to localtime re-entrantly.

"HAS_MKTIME"

This symbol, if defined, indicates that the "mktime" routine is available.

"HAS_MKTIME64"

This symbol, if defined, indicates that the "mktime64" () routine is available to do the 64bit variant of mktime ()

"HAS_NANOSLEEP"

This symbol, if defined, indicates that the "nanosleep" system call is available to sleep with 1E-9 sec accuracy.

"HAS_SETITIMER"

This symbol, if defined, indicates that the "setitimer" routine is available to set interval timers.

"HAS_STRFTIME"

This symbol, if defined, indicates that the "strftime" routine is available to do time

formatting.

"HAS_TIME"

This symbol, if defined, indicates that the "time()" routine exists.

"HAS_TIMEGM"

This symbol, if defined, indicates that the "timegm" routine is available to do the opposite of gmtime ()

"HAS_TIMES"

This symbol, if defined, indicates that the "times()" routine exists. Note that this became obsolete on some systems ("SUNOS"), which now use "getrusage()". It may be necessary to include sys/times.h.

"HAS_TM_TM_GMTOFF"

This symbol, if defined, indicates to the C program that the "struct tm" has a "tm_gmtoff" field.

"HAS_TM_TM_ZONE"

This symbol, if defined, indicates to the C program that the "struct tm" has a "tm_zone" field.

"HAS_TZNAME"

This symbol, if defined, indicates that the "tzname[]" array is available to access timezone names.

"HAS_USLEEP"

This symbol, if defined, indicates that the "usleep" routine is available to let the process sleep on a sub-second accuracy.

"HAS_USLEEP_PROTO"

This symbol, if defined, indicates that the system provides a prototype for the "usleep()" function. Otherwise, it is up to the program to supply one. A good guess is

```
 extern int usleep(useconds_t);
```

"I_TIME"

This symbol is always defined, and indicates to the C program that it should include time.h.

```
 #ifdef I_TIME
     #include <time.h>
 #endif
```

"I_UTIME"

This symbol, if defined, indicates to the C program that it should include utime.h.

```
#ifdef I_UTIME
    #include <utime.h>
#endif
```

"LOCALTIME_MAX"

This symbol contains the maximum value for the "time_t" offset that the system function localtime () accepts, and defaults to 0

"LOCALTIME_MIN"

This symbol contains the minimum value for the "time_t" offset that the system function localtime () accepts, and defaults to 0

"LOCALTIME_R_NEEDS_TZSET"

Many libc's "localtime_r" implementations do not call tzset, making them differ from "localtime()", and making timezone changes using $"ENV"{TZ} without explicitly calling tzset impossible. This symbol makes us call tzset before "localtime_r"

"LOCALTIME_R_PROTO"

This symbol encodes the prototype of "localtime_r". It is zero if "d_localtime_r" is undef, and one of the "REENTRANT_PROTO_T_ABC" macros of reentr.h if "d_localtime_r" is defined.

"L_R_TZSET"

If "localtime_r()" needs tzset, it is defined in this define

"mini_mktime"

normalise "struct?tm" values without the localtime() semantics (and overhead) of mktime().

```
void  mini_mktime(struct tm *ptm)
```

"my_strftime"

strftime(), but with a different API so that the return value is a pointer to the formatted result (which MUST be arranged to be FREED BY THE CALLER). This allows this function to increase the buffer size as needed, so that the caller doesn't have to worry about that.

Note that yday and wday effectively are ignored by this function, as mini_mktime() overwrites them

Also note that this is always executed in the underlying locale of the program, giving

localized results.

NOTE: "my_strftime" must be explicitly called as "Perl_my_strftime" with an "aTHX_" parameter.

```
 char *  Perl_my_strftime(pTHX_ const char *fmt, int sec, int min,
                          int hour, int mday, int mon, int year,
                          int wday, int yday, int isdst)
```

Typedef names

"DB_Hash_t"

This symbol contains the type of the prefix structure element in the db.h header file. In older versions of DB, it was int, while in newer ones it is "size_t".

"DB_Prefix_t"

This symbol contains the type of the prefix structure element in the db.h header file. In older versions of DB, it was int, while in newer ones it is "u_int32_t".

"Direntry_t"

This symbol is set to "'struct direct'" or "'struct dirent'" depending on whether dirent is available or not. You should use this pseudo type to portably declare your directory entries.

"Fpos_t"

This symbol holds the type used to declare file positions in libc. It can be "fpos_t", long, uint, etc... It may be necessary to include sys/types.h to get any typedef'ed information.

"Free_t"

This variable contains the return type of "free()". It is usually void, but occasionally int.

"Gid_t"

This symbol holds the return type of "getgid()" and the type of argument to "setrgid()" and related functions. Typically, it is the type of group ids in the kernel. It can be int, ushort, "gid_t", etc... It may be necessary to include sys/types.h to get any typedef'ed information.

"Gid_t_f"

This symbol defines the format string used for printing a "Gid_t".

"Gid_t_sign"

This symbol holds the signedness of a "Gid_t". 1 for unsigned, -1 for signed.

"Gid_t_size"

This symbol holds the size of a "Gid_t" in bytes.

"Groups_t"

This symbol holds the type used for the second argument to "getgroups()" and "setgroups()". Usually, this is the same as gidtype ("gid_t") , but sometimes it isn't. It can be int, ushort, "gid_t", etc... It may be necessary to include sys/types.h to get any typedef'ed information. This is only required if you have "getgroups()" or "setgroups()"..

"Malloc_t"

This symbol is the type of pointer returned by malloc and realloc.

"Mmap_t"

This symbol holds the return type of the "mmap()" system call (and simultaneously the type of the first argument). Usually set to 'void *' or "'caddr_t"'.

"Mode_t"

This symbol holds the type used to declare file modes for systems calls. It is usually "mode_t", but may be int or unsigned short. It may be necessary to include sys/types.h to get any typedef'ed information.

"Netdb_hlen_t"

This symbol holds the type used for the 2nd argument to "gethostbyaddr()".

"Netdb_host_t"

This symbol holds the type used for the 1st argument to "gethostbyaddr()".

"Netdb_name_t"

This symbol holds the type used for the argument to "gethostbyname()".

"Netdb_net_t"

This symbol holds the type used for the 1st argument to "getnetbyaddr()".

"Off_t"

This symbol holds the type used to declare offsets in the kernel. It can be int, long, "off_t", etc... It may be necessary to include sys/types.h to get any typedef'ed information.

"Off_t_size"

This symbol holds the number of bytes used by the "Off_t".

"Pid_t"

This symbol holds the type used to declare process ids in the kernel. It can be int,

uint, "pid_t", etc... It may be necessary to include sys/types.h to get any typedef'ed information.

"Rand_seed_t"

This symbol defines the type of the argument of the random seed function.

"Select_fd_set_t"

This symbol holds the type used for the 2nd, 3rd, and 4th arguments to select. Usually, this is '"fd_set" *', if "HAS_FD_SET" is defined, and 'int *' otherwise. This is only useful if you have "select()", of course.

"Shmat_t"

This symbol holds the return type of the "shmat()" system call. Usually set to 'void *' or 'char *'.

"Signal_t"

This symbol's value is either "void" or "int", corresponding to the appropriate return type of a signal handler. Thus, you can declare a signal handler using ""Signal_t" (*handler)()", and define the handler using ""Signal_t" "handler(sig)"".

"Size_t"

This symbol holds the type used to declare length parameters for string functions. It is usually "size_t", but may be unsigned long, int, etc. It may be necessary to include sys/types.h to get any typedef'ed information.

"Size_t_size"

This symbol holds the size of a "Size_t" in bytes.

"Sock_size_t"

This symbol holds the type used for the size argument of various socket calls (just the base type, not the pointer-to).

"SSize_t"

This symbol holds the type used by functions that return a count of bytes or an error condition. It must be a signed type. It is usually "ssize_t", but may be long or int, etc. It may be necessary to include sys/types.h or unistd.h to get any typedef'ed information. We will pick a type such that "sizeof(SSize_t)" == "sizeof(Size_t)".

"Time_t"

This symbol holds the type returned by "time()". It can be long, or "time_t" on "BSD" sites (in which case sys/types.h should be included).

"Uid_t"

This symbol holds the type used to declare user ids in the kernel. It can be int, ushort, "uid_t", etc... It may be necessary to include sys/types.h to get any typedef'ed information.

"Uid_t_f"

This symbol defines the format string used for printing a "Uid_t".

"Uid_t_sign"

This symbol holds the signedness of a "Uid_t". 1 for unsigned, -1 for signed.

"Uid_t_size"

This symbol holds the size of a "Uid_t" in bytes.

Unicode Support

"Unicode Support" in perlguts has an introduction to this API.

See also "Character classification", "Character case changing", and "String Handling".

Various functions outside this section also work specially with Unicode. Search for the string "utf8" in this document.

"BOM_UTF8"

This is a macro that evaluates to a string constant of the UTF-8 bytes that define the Unicode BYTE ORDER MARK (U+FEFF) for the platform that perl is compiled on. This allows code to use a mnemonic for this character that works on both ASCII and EBCDIC platforms. "sizeof(BOM_UTF8)?-?1" can be used to get its length in bytes.

"bytes_cmp_utf8"

Compares the sequence of characters (stored as octets) in "b", "blen" with the sequence of characters (stored as UTF-8) in "u", "ulen". Returns 0 if they are equal, -1 or -2 if the first string is less than the second string, +1 or +2 if the first string is greater than the second string.

-1 or +1 is returned if the shorter string was identical to the start of the longer string. -2 or +2 is returned if there was a difference between characters within the strings.

```
 int  bytes_cmp_utf8(const U8 *b, STRLEN blen, const U8 *u,
                     STRLEN ulen)
```

"bytes_from_utf8"

NOTE: "bytes_from_utf8" is experimental and may change or be removed without notice.

Converts a potentially UTF-8 encoded string "s" of length *lenp into native byte

encoding. On input, the boolean *is_utf8p gives whether or not "s" is actually encoded in UTF-8.

Unlike "utf8_to_bytes" but like "bytes_to_utf8", this is non-destructive of the input string.

Do nothing if *is_utf8p is 0, or if there are code points in the string not expressible in native byte encoding. In these cases, *is_utf8p and *lenp are unchanged, and the return value is the original "s".

Otherwise, *is_utf8p is set to 0, and the return value is a pointer to a newly created string containing a downgraded copy of "s", and whose length is returned in *lenp, updated. The new string is "NUL"-terminated. The caller is responsible for arranging for the memory used by this string to get freed.

Upon successful return, the number of variants in the string can be computed by having saved the value of *lenp before the call, and subtracting the after-call value of *lenp from it.

U8* bytes_from_utf8(const U8 *s, STRLEN *lenp, bool *is_utf8p)

"bytes_to_utf8"

NOTE: "bytes_to_utf8" is experimental and may change or be removed without notice.

Converts a string "s" of length *lenp bytes from the native encoding into UTF-8.

Returns a pointer to the newly-created string, and sets *lenp to reflect the new length in bytes. The caller is responsible for arranging for the memory used by this string to get freed.

Upon successful return, the number of variants in the string can be computed by having saved the value of *lenp before the call, and subtracting it from the after-call value of *lenp.

A "NUL" character will be written after the end of the string.

If you want to convert to UTF-8 from encodings other than the native (Latin1 or EBCDIC), see "sv_recode_to_utf8"().

U8* bytes_to_utf8(const U8 *s, STRLEN *lenp)

"DO_UTF8"

Returns a bool giving whether or not the PV in "sv" is to be treated as being encoded in UTF-8.

You should use this after a call to "SvPV()" or one of its variants, in case any call to string overloading updates the internal UTF-8 encoding flag.

bool DO_UTF8(SV* sv)

"foldEQ_utf8"

Returns true if the leading portions of the strings "s1" and "s2" (either or both of which may be in UTF-8) are the same case-insensitively; false otherwise. How far into the strings to compare is determined by other input parameters.

If "u1" is true, the string "s1" is assumed to be in UTF-8-encoded Unicode; otherwise it is assumed to be in native 8-bit encoding. Correspondingly for "u2" with respect to "s2".

If the byte length "l1" is non-zero, it says how far into "s1" to check for fold equality. In other words, "s1"+"l1" will be used as a goal to reach. The scan will not be considered to be a match unless the goal is reached, and scanning won't continue past that goal. Correspondingly for "l2" with respect to "s2".

If "pe1" is non-"NULL" and the pointer it points to is not "NULL", that pointer is considered an end pointer to the position 1 byte past the maximum point in "s1" beyond which scanning will not continue under any circumstances. (This routine assumes that UTF-8 encoded input strings are not malformed; malformed input can cause it to read past "pe1"). This means that if both "l1" and "pe1" are specified, and "pe1" is less than "s1"+"l1", the match will never be successful because it can never get as far as its goal (and in fact is asserted against). Correspondingly for "pe2" with respect to "s2".

At least one of "s1" and "s2" must have a goal (at least one of "l1" and "l2" must be non-zero), and if both do, both have to be reached for a successful match. Also, if the fold of a character is multiple characters, all of them must be matched (see tr21 reference below for 'folding').

Upon a successful match, if "pe1" is non-"NULL", it will be set to point to the beginning of the next character of "s1" beyond what was matched. Correspondingly for "pe2" and "s2".

For case-insensitiveness, the "casefolding" of Unicode is used instead of upper/lowercasing both the characters, see <https://www.unicode.org/unicode/reports/tr21/> (Case Mappings).

I32 foldEQ_utf8(const char *s1, char **pe1, UV l1, bool u1,
const char *s2, char **pe2, UV l2, bool u2)

"is_ascii_string"

This is a misleadingly-named synonym for "is_utf8_invariant_string". On ASCII-ish platforms, the name isn't misleading: the ASCII-range characters are exactly the UTF-8 invariants. But EBCDIC machines have more invariants than just the ASCII characters, so "is_utf8_invariant_string" is preferred.

bool is_ascii_string(const U8* const s, STRLEN len)

"is_c9strict_utf8_string"

Returns TRUE if the first "len" bytes of string "s" form a valid UTF-8-encoded string that conforms to Unicode Corrigendum #9 <http://www.unicode.org/versions/corrigendum9.html>; otherwise it returns FALSE. If "len" is 0, it will be calculated using strlen(s) (which means if you use this option, that "s" can't have embedded "NUL" characters and has to have a terminating "NUL" byte). Note that all characters being ASCII constitute 'a valid UTF-8 string'.

This function returns FALSE for strings containing any code points above the Unicode max of 0x10FFFF or surrogate code points, but accepts non-character code points per Corrigendum #9 <http://www.unicode.org/versions/corrigendum9.html>.

See also "is_utf8_invariant_string", "is_utf8_invariant_string_loc", "is_utf8_string", "is_utf8_string_flags", "is_utf8_string_loc", "is_utf8_string_loc_flags", "is_utf8_string_loclen", "is_utf8_string_loclen_flags", "is_utf8_fixed_width_buf_flags", "is_utf8_fixed_width_buf_loc_flags", "is_utf8_fixed_width_buf_loclen_flags", "is_strict_utf8_string", "is_strict_utf8_string_loc", "is_strict_utf8_string_loclen", "is_c9strict_utf8_string_loc", and "is_c9strict_utf8_string_loclen".

bool is_c9strict_utf8_string(const U8 *s, STRLEN len)

"is_c9strict_utf8_string_loc"

Like "is_c9strict_utf8_string" but stores the location of the failure (in the case of "utf8ness failure") or the location "s"+"len" (in the case of "utf8ness success") in the "ep" pointer.

See also "is_c9strict_utf8_string_loclen".

bool is_c9strict_utf8_string_loc(const U8 *s, STRLEN len,
const U8 **ep)

"is_c9strict_utf8_string_loclen"

Like "is_c9strict_utf8_string" but stores the location of the failure (in the case of "utf8ness failure") or the location "s"+"len" (in the case of "utf8ness success") in

the "ep" pointer, and the number of UTF-8 encoded characters in the "el" pointer.

See also "is_c9strict_utf8_string_loc".

```
 bool  is_c9strict_utf8_string_loclen(const U8 *s, STRLEN len,
                                      const U8 **ep, STRLEN *el)
```

"isC9_STRICT_UTF8_CHAR"

Evaluates to non-zero if the first few bytes of the string starting at "s" and looking no further than "e?-?1" are well-formed UTF-8 that represents some Unicode non-surrogate code point; otherwise it evaluates to 0. If non-zero, the value gives how many bytes starting at "s" comprise the code point's representation. Any bytes remaining before "e", but beyond the ones needed to form the first code point in "s", are not examined.

The largest acceptable code point is the Unicode maximum 0x10FFFF. This differs from "isSTRICT_UTF8_CHAR" only in that it accepts non-character code points. This corresponds to Unicode Corrigendum #9 <http://www.unicode.org/versions/corrigendum9.html>. which said that non-character code points are merely discouraged rather than completely forbidden in open interchange. See "Noncharacter code points" in perlunicode.

Use "isUTF8_CHAR" to check for Perl's extended UTF-8; and "isUTF8_CHAR_flags" for a more customized definition.

Use "is_c9strict_utf8_string", "is_c9strict_utf8_string_loc", and "is_c9strict_utf8_string_loclen" to check entire strings.

```
 Size_t  isC9_STRICT_UTF8_CHAR(const U8 * const s0,
                               const U8 * const e)
```

"is_invariant_string"

This is a somewhat misleadingly-named synonym for "is_utf8_invariant_string". "is_utf8_invariant_string" is preferred, as it indicates under what conditions the string is invariant.

```
 bool  is_invariant_string(const U8* const s, STRLEN len)
```

"isSTRICT_UTF8_CHAR"

Evaluates to non-zero if the first few bytes of the string starting at "s" and looking no further than "e?-?1" are well-formed UTF-8 that represents some Unicode code point completely acceptable for open interchange between all applications; otherwise it evaluates to 0. If non-zero, the value gives how many bytes starting at "s" comprise

the code point's representation. Any bytes remaining before "e", but beyond the ones needed to form the first code point in "s", are not examined.

The largest acceptable code point is the Unicode maximum 0x10FFFF, and must not be a surrogate nor a non-character code point. Thus this excludes any code point from Perl's extended UTF-8.

This is used to efficiently decide if the next few bytes in "s" is legal Unicode-acceptable UTF-8 for a single character.

Use "isC9_STRICT_UTF8_CHAR" to use the Unicode Corrigendum #9 <http://www.unicode.org/versions/corrigendum9.html> definition of allowable code points; "isUTF8_CHAR" to check for Perl's extended UTF-8; and "isUTF8_CHAR_flags" for a more customized definition.

Use "is_strict_utf8_string", "is_strict_utf8_string_loc", and "is_strict_utf8_string_loclen" to check entire strings.

```
 Size_t  isSTRICT_UTF8_CHAR(const U8 * const s0,
                            const U8 * const e)
```

"is_strict_utf8_string"

Returns TRUE if the first "len" bytes of string "s" form a valid UTF-8-encoded string that is fully interchangeable by any application using Unicode rules; otherwise it returns FALSE. If "len" is 0, it will be calculated using strlen(s) (which means if you use this option, that "s" can't have embedded "NUL" characters and has to have a terminating "NUL" byte). Note that all characters being ASCII constitute 'a valid UTF-8 string'.

This function returns FALSE for strings containing any code points above the Unicode max of 0x10FFFF, surrogate code points, or non-character code points.

See also "is_utf8_invariant_string", "is_utf8_invariant_string_loc", "is_utf8_string", "is_utf8_string_flags", "is_utf8_string_loc", "is_utf8_string_loc_flags", "is_utf8_string_loclen", "is_utf8_string_loclen_flags", "is_utf8_fixed_width_buf_flags", "is_utf8_fixed_width_buf_loc_flags", "is_utf8_fixed_width_buf_loclen_flags", "is_strict_utf8_string_loc", "is_strict_utf8_string_loclen", "is_c9strict_utf8_string", "is_c9strict_utf8_string_loc", and "is_c9strict_utf8_string_loclen".

```
 bool  is_strict_utf8_string(const U8 *s, STRLEN len)
```

"is_strict_utf8_string_loc"

Like "is_strict_utf8_string" but stores the location of the failure (in the case of "utf8ness failure") or the location "s"+"len" (in the case of "utf8ness success") in the "ep" pointer.

See also "is_strict_utf8_string_loclen".

```
 bool  is_strict_utf8_string_loc(const U8 *s, STRLEN len,
                                 const U8 **ep)
```

"is_strict_utf8_string_loclen"

Like "is_strict_utf8_string" but stores the location of the failure (in the case of "utf8ness failure") or the location "s"+"len" (in the case of "utf8ness success") in the "ep" pointer, and the number of UTF-8 encoded characters in the "el" pointer.

See also "is_strict_utf8_string_loc".

```
 bool  is_strict_utf8_string_loclen(const U8 *s, STRLEN len,
                                    const U8 **ep, STRLEN *el)
```

"is_utf8_char"

"DEPRECATED!" It is planned to remove "is_utf8_char" from a future release of Perl. Do not use it for new code; remove it from existing code.

Tests if some arbitrary number of bytes begins in a valid UTF-8 character. Note that an INVARIANT (i.e. ASCII on non-EBCDIC machines) character is a valid UTF-8 character. The actual number of bytes in the UTF-8 character will be returned if it is valid, otherwise 0.

This function is deprecated due to the possibility that malformed input could cause reading beyond the end of the input buffer. Use "isUTF8_CHAR" instead.

```
 STRLEN  is_utf8_char(const U8 *s)
```

"is_utf8_char_buf"

This is identical to the macro "isUTF8_CHAR" in perlapi.

```
 STRLEN  is_utf8_char_buf(const U8 *buf, const U8 *buf_end)
```

"is_utf8_fixed_width_buf_flags"

Returns TRUE if the fixed-width buffer starting at "s" with length "len" is entirely valid UTF-8, subject to the restrictions given by "flags"; otherwise it returns FALSE.

If "flags" is 0, any well-formed UTF-8, as extended by Perl, is accepted without restriction. If the final few bytes of the buffer do not form a complete code point, this will return TRUE anyway, provided that "is_utf8_valid_partial_char_flags" returns TRUE for them.

If "flags" in non-zero, it can be any combination of the "UTF8_DISALLOW_foo" flags accepted by "utf8n_to_uvchr", and with the same meanings.

This function differs from "is_utf8_string_flags" only in that the latter returns FALSE if the final few bytes of the string don't form a complete code point.

```
 bool  is_utf8_fixed_width_buf_flags(const U8 * const s,
                                     STRLEN len, const U32 flags)
```

"is_utf8_fixed_width_buf_loclen_flags"

Like "is_utf8_fixed_width_buf_loc_flags" but stores the number of complete, valid characters found in the "el" pointer.

```
 bool  is_utf8_fixed_width_buf_loclen_flags(const U8 * const s,
                                            STRLEN len,
                                            const U8 **ep,
                                            STRLEN *el,
                                            const U32 flags)
```

"is_utf8_fixed_width_buf_loc_flags"

Like "is_utf8_fixed_width_buf_flags" but stores the location of the failure in the "ep" pointer. If the function returns TRUE, *ep will point to the beginning of any partial character at the end of the buffer; if there is no partial character *ep will contain "s"+"len".

See also "is_utf8_fixed_width_buf_loclen_flags".

```
 bool  is_utf8_fixed_width_buf_loc_flags(const U8 * const s,
                                         STRLEN len, const U8 **ep,
                                         const U32 flags)
```

"is_utf8_invariant_string"

Returns TRUE if the first "len" bytes of the string "s" are the same regardless of the UTF-8 encoding of the string (or UTF-EBCDIC encoding on EBCDIC machines); otherwise it returns FALSE. That is, it returns TRUE if they are UTF-8 invariant. On ASCII-ish machines, all the ASCII characters and only the ASCII characters fit this definition. On EBCDIC machines, the ASCII-range characters are invariant, but so also are the C1 controls.

If "len" is 0, it will be calculated using strlen(s), (which means if you use this option, that "s" can't have embedded "NUL" characters and has to have a terminating "NUL" byte).

See also "is_utf8_string", "is_utf8_string_flags", "is_utf8_string_loc",
"is_utf8_string_loc_flags", "is_utf8_string_loclen", "is_utf8_string_loclen_flags",
"is_utf8_fixed_width_buf_flags", "is_utf8_fixed_width_buf_loc_flags",
"is_utf8_fixed_width_buf_loclen_flags", "is_strict_utf8_string",
"is_strict_utf8_string_loc", "is_strict_utf8_string_loclen",
"is_c9strict_utf8_string", "is_c9strict_utf8_string_loc", and
"is_c9strict_utf8_string_loclen".

bool is_utf8_invariant_string(const U8* const s, STRLEN len)

"is_utf8_invariant_string_loc"

Like "is_utf8_invariant_string" but upon failure, stores the location of the first
UTF-8 variant character in the "ep" pointer; if all characters are UTF-8 invariant,
this function does not change the contents of *ep.

bool is_utf8_invariant_string_loc(const U8* const s, STRLEN len,
const U8 ** ep)

"is_utf8_string"

Returns TRUE if the first "len" bytes of string "s" form a valid Perl-extended-UTF-8
string; returns FALSE otherwise. If "len" is 0, it will be calculated using strlen(s)
(which means if you use this option, that "s" can't have embedded "NUL" characters and
has to have a terminating "NUL" byte). Note that all characters being ASCII
constitute 'a valid UTF-8 string'.

This function considers Perl's extended UTF-8 to be valid. That means that code
points above Unicode, surrogates, and non-character code points are considered valid
by this function. Use "is_strict_utf8_string", "is_c9strict_utf8_string", or
"is_utf8_string_flags" to restrict what code points are considered valid.

See also "is_utf8_invariant_string", "is_utf8_invariant_string_loc",
"is_utf8_string_loc", "is_utf8_string_loclen", "is_utf8_fixed_width_buf_flags",
"is_utf8_fixed_width_buf_loc_flags", "is_utf8_fixed_width_buf_loclen_flags",

bool is_utf8_string(const U8 *s, STRLEN len)

"is_utf8_string_flags"

Returns TRUE if the first "len" bytes of string "s" form a valid UTF-8 string, subject
to the restrictions imposed by "flags"; returns FALSE otherwise. If "len" is 0, it
will be calculated using strlen(s) (which means if you use this option, that "s" can't
have embedded "NUL" characters and has to have a terminating "NUL" byte). Note that

all characters being ASCII constitute 'a valid UTF-8 string'.

If "flags" is 0, this gives the same results as "is_utf8_string"; if "flags" is "UTF8_DISALLOW_ILLEGAL_INTERCHANGE", this gives the same results as "is_strict_utf8_string"; and if "flags" is "UTF8_DISALLOW_ILLEGAL_C9_INTERCHANGE", this gives the same results as "is_c9strict_utf8_string". Otherwise "flags" may be any combination of the "UTF8_DISALLOW_foo" flags understood by "utf8n_to_uvchr", with the same meanings.

See also "is_utf8_invariant_string", "is_utf8_invariant_string_loc", "is_utf8_string", "is_utf8_string_loc", "is_utf8_string_loc_flags", "is_utf8_string_loclen", "is_utf8_string_loclen_flags", "is_utf8_fixed_width_buf_flags", "is_utf8_fixed_width_buf_loc_flags", "is_utf8_fixed_width_buf_loclen_flags", "is_strict_utf8_string", "is_strict_utf8_string_loc", "is_strict_utf8_string_loclen", "is_c9strict_utf8_string", "is_c9strict_utf8_string_loc", and "is_c9strict_utf8_string_loclen".

```
 bool  is_utf8_string_flags(const U8 *s, STRLEN len,
                            const U32 flags)
```

"is_utf8_string_loc"

Like "is_utf8_string" but stores the location of the failure (in the case of "utf8ness failure") or the location "s"+"len" (in the case of "utf8ness success") in the "ep" pointer.

See also "is_utf8_string_loclen".

```
 bool  is_utf8_string_loc(const U8 *s, const STRLEN len,
                          const U8 **ep)
```

"is_utf8_string_loclen"

Like "is_utf8_string" but stores the location of the failure (in the case of "utf8ness failure") or the location "s"+"len" (in the case of "utf8ness success") in the "ep" pointer, and the number of UTF-8 encoded characters in the "el" pointer.

See also "is_utf8_string_loc".

```
 bool  is_utf8_string_loclen(const U8 *s, STRLEN len,
                             const U8 **ep, STRLEN *el)
```

"is_utf8_string_loclen_flags"

Like "is_utf8_string_flags" but stores the location of the failure (in the case of "utf8ness failure") or the location "s"+"len" (in the case of "utf8ness success") in

the "ep" pointer, and the number of UTF-8 encoded characters in the "el" pointer.

See also "is_utf8_string_loc_flags".

```
 bool  is_utf8_string_loclen_flags(const U8 *s, STRLEN len,
                                   const U8 **ep, STRLEN *el,
                                   const U32 flags)
```

"is_utf8_string_loc_flags"

Like "is_utf8_string_flags" but stores the location of the failure (in the case of "utf8ness failure") or the location "s"+"len" (in the case of "utf8ness success") in the "ep" pointer.

See also "is_utf8_string_loclen_flags".

```
 bool  is_utf8_string_loc_flags(const U8 *s, STRLEN len,
                                const U8 **ep, const U32 flags)
```

"is_utf8_valid_partial_char"

Returns 0 if the sequence of bytes starting at "s" and looking no further than "e?-?1" is the UTF-8 encoding, as extended by Perl, for one or more code points. Otherwise, it returns 1 if there exists at least one non-empty sequence of bytes that when appended to sequence "s", starting at position "e" causes the entire sequence to be the well-formed UTF-8 of some code point; otherwise returns 0.

In other words this returns TRUE if "s" points to a partial UTF-8-encoded code point. This is useful when a fixed-length buffer is being tested for being well-formed UTF-8, but the final few bytes in it don't comprise a full character; that is, it is split somewhere in the middle of the final code point's UTF-8 representation. (Presumably when the buffer is refreshed with the next chunk of data, the new first bytes will complete the partial code point.) This function is used to verify that the final bytes in the current buffer are in fact the legal beginning of some code point, so that if they aren't, the failure can be signalled without having to wait for the next read.

```
 bool  is_utf8_valid_partial_char(const U8 * const s,
                                  const U8 * const e)
```

"is_utf8_valid_partial_char_flags"

Like "is_utf8_valid_partial_char", it returns a boolean giving whether or not the input is a valid UTF-8 encoded partial character, but it takes an extra parameter, "flags", which can further restrict which code points are considered valid.

If "flags" is 0, this behaves identically to "is_utf8_valid_partial_char". Otherwise "flags" can be any combination of the "UTF8_DISALLOW_foo" flags accepted by "utf8n_to_uvchr". If there is any sequence of bytes that can complete the input partial character in such a way that a non-prohibited character is formed, the function returns TRUE; otherwise FALSE. Non character code points cannot be determined based on partial character input. But many of the other possible excluded types can be determined from just the first one or two bytes.

```
bool  is_utf8_valid_partial_char_flags(const U8 * const s,
                                       const U8 * const e,
                                       const U32 flags)
```

"isUTF8_CHAR"

Evaluates to non-zero if the first few bytes of the string starting at "s" and looking no further than "e?-?1" are well-formed UTF-8, as extended by Perl, that represents some code point; otherwise it evaluates to 0. If non-zero, the value gives how many bytes starting at "s" comprise the code point's representation. Any bytes remaining before "e", but beyond the ones needed to form the first code point in "s", are not examined.

The code point can be any that will fit in an IV on this machine, using Perl's extension to official UTF-8 to represent those higher than the Unicode maximum of 0x10FFFF. That means that this macro is used to efficiently decide if the next few bytes in "s" is legal UTF-8 for a single character.

Use "isSTRICT_UTF8_CHAR" to restrict the acceptable code points to those defined by Unicode to be fully interchangeable across applications; "isC9_STRICT_UTF8_CHAR" to use the Unicode Corrigendum #9 <http://www.unicode.org/versions/corrigendum9.html> definition of allowable code points; and "isUTF8_CHAR_flags" for a more customized definition.

Use "is_utf8_string", "is_utf8_string_loc", and "is_utf8_string_loclen" to check entire strings.

Note also that a UTF-8 "invariant" character (i.e. ASCII on non-EBCDIC machines) is a valid UTF-8 character.

```
Size_t  isUTF8_CHAR(const U8 * const s0, const U8 * const e)
```

"isUTF8_CHAR_flags"

Evaluates to non-zero if the first few bytes of the string starting at "s" and looking

no further than "e?-?1" are well-formed UTF-8, as extended by Perl, that represents some code point, subject to the restrictions given by "flags"; otherwise it evaluates to 0. If non-zero, the value gives how many bytes starting at "s" comprise the code point's representation. Any bytes remaining before "e", but beyond the ones needed to form the first code point in "s", are not examined.

If "flags" is 0, this gives the same results as "isUTF8_CHAR"; if "flags" is "UTF8_DISALLOW_ILLEGAL_INTERCHANGE", this gives the same results as "isSTRICT_UTF8_CHAR"; and if "flags" is "UTF8_DISALLOW_ILLEGAL_C9_INTERCHANGE", this gives the same results as "isC9_STRICT_UTF8_CHAR". Otherwise "flags" may be any combination of the "UTF8_DISALLOW_foo" flags understood by "utf8n_to_uvchr", with the same meanings.

The three alternative macros are for the most commonly needed validations; they are likely to run somewhat faster than this more general one, as they can be inlined into your code.

Use "is_utf8_string_flags", "is_utf8_string_loc_flags", and "is_utf8_string_loclen_flags" to check entire strings.

```
STRLEN  isUTF8_CHAR_flags(const U8 *s, const U8 *e,
                          const U32 flags)
```

"LATIN1_TO_NATIVE"

Returns the native equivalent of the input Latin-1 code point (including ASCII and control characters) given by "ch". Thus, "LATIN1_TO_NATIVE(66)" on EBCDIC platforms returns 194. These each represent the character "B" on their respective platforms.

On ASCII platforms no conversion is needed, so this macro expands to just its input, adding no time nor space requirements to the implementation.

For conversion of code points potentially larger than will fit in a character, use "UNI_TO_NATIVE".

```
U8  LATIN1_TO_NATIVE(U8 ch)
```

"NATIVE_TO_LATIN1"

Returns the Latin-1 (including ASCII and control characters) equivalent of the input native code point given by "ch". Thus, "NATIVE_TO_LATIN1(193)" on EBCDIC platforms returns 65. These each represent the character "A" on their respective platforms. On ASCII platforms no conversion is needed, so this macro expands to just its input, adding no time nor space requirements to the implementation.

For conversion of code points potentially larger than will fit in a character, use "NATIVE_TO_UNI".

U8 NATIVE_TO_LATIN1(U8 ch)

"NATIVE_TO_UNI"

Returns the Unicode equivalent of the input native code point given by "ch". Thus, "NATIVE_TO_UNI(195)" on EBCDIC platforms returns 67. These each represent the character "C" on their respective platforms. On ASCII platforms no conversion is needed, so this macro expands to just its input, adding no time nor space requirements to the implementation.

UV NATIVE_TO_UNI(UV ch)

"pad_compname_type"

"DEPRECATED!" It is planned to remove "pad_compname_type" from a future release of Perl. Do not use it for new code; remove it from existing code.

Looks up the type of the lexical variable at position "po" in the currently-compiling pad. If the variable is typed, the stash of the class to which it is typed is returned. If not, "NULL" is returned.

Use ""PAD_COMPNAME_TYPE"" in perlintern instead.

HV* pad_compname_type(const PADOFFSET po)

"pv_uni_display"

Build to the scalar "dsv" a displayable version of the UTF-8 encoded string "spv", length "len", the displayable version being at most "pvlim" bytes long (if longer, the rest is truncated and "..." will be appended).

The "flags" argument can have "UNI_DISPLAY_ISPRINT" set to display "isPRINT()"able characters as themselves, "UNI_DISPLAY_BACKSLASH" to display the "\\[nrfta\\]" as the backslashed versions (like "\n") ("UNI_DISPLAY_BACKSLASH" is preferred over "UNI_DISPLAY_ISPRINT" for "\\"). "UNI_DISPLAY_QQ" (and its alias "UNI_DISPLAY_REGEX") have both "UNI_DISPLAY_BACKSLASH" and "UNI_DISPLAY_ISPRINT" turned on.

Additionally, there is now "UNI_DISPLAY_BACKSPACE" which allows "\b" for a backspace, but only when "UNI_DISPLAY_BACKSLASH" also is set.

The pointer to the PV of the "dsv" is returned.

See also "sv_uni_display".

char* pv_uni_display(SV *dsv, const U8 *spv, STRLEN len,
STRLEN pvlim, UV flags)

"REPLACEMENT_CHARACTER_UTF8"

This is a macro that evaluates to a string constant of the UTF-8 bytes that define the Unicode REPLACEMENT CHARACTER (U+FFFD) for the platform that perl is compiled on. This allows code to use a mnemonic for this character that works on both ASCII and EBCDIC platforms. "sizeof(REPLACEMENT_CHARACTER_UTF8)?-?1" can be used to get its length in bytes.

"sv_cat_decode"

"encoding" is assumed to be an "Encode" object, the PV of "ssv" is assumed to be octets in that encoding and decoding the input starts from the position which "(PV?+?*offset)" pointed to. "dsv" will be concatenated with the decoded UTF-8 string from "ssv". Decoding will terminate when the string "tstr" appears in decoding output or the input ends on the PV of "ssv". The value which "offset" points will be modified to the last input position on "ssv".

Returns TRUE if the terminator was found, else returns FALSE.

```
bool  sv_cat_decode(SV* dsv, SV *encoding, SV *ssv, int *offset,
                    char* tstr, int tlen)
```

"sv_recode_to_utf8"

"encoding" is assumed to be an "Encode" object, on entry the PV of "sv" is assumed to be octets in that encoding, and "sv" will be converted into Unicode (and UTF-8). If "sv" already is UTF-8 (or if it is not "POK"), or if "encoding" is not a reference, nothing is done to "sv". If "encoding" is not an "Encode::XS" Encoding object, bad things will happen. (See cpan/Encode/encoding.pm and Encode.)

The PV of "sv" is returned.

```
char*  sv_recode_to_utf8(SV* sv, SV *encoding)
```

"sv_uni_display"

Build to the scalar "dsv" a displayable version of the scalar "sv", the displayable version being at most "pvlim" bytes long (if longer, the rest is truncated and "..." will be appended).

The "flags" argument is as in "pv_uni_display"().

The pointer to the PV of the "dsv" is returned.

```
char*  sv_uni_display(SV *dsv, SV *ssv, STRLEN pvlim, UV flags)
```

"UNICODE_REPLACEMENT"

Evaluates to 0xFFFD, the code point of the Unicode REPLACEMENT CHARACTER

"UNI_TO_NATIVE"

Returns the native equivalent of the input Unicode code point given by "ch". Thus, "UNI_TO_NATIVE(68)" on EBCDIC platforms returns 196. These each represent the character "D" on their respective platforms. On ASCII platforms no conversion is needed, so this macro expands to just its input, adding no time nor space requirements to the implementation.

UV UNI_TO_NATIVE(UV ch)

"utf8n_to_uvchr"

THIS FUNCTION SHOULD BE USED IN ONLY VERY SPECIALIZED CIRCUMSTANCES. Most code should use "utf8_to_uvchr_buf"() rather than call this directly.

Bottom level UTF-8 decode routine. Returns the native code point value of the first character in the string "s", which is assumed to be in UTF-8 (or UTF-EBCDIC) encoding, and no longer than "curlen" bytes; *retlen (if "retlen" isn't NULL) will be set to the length, in bytes, of that character.

The value of "flags" determines the behavior when "s" does not point to a well-formed UTF-8 character. If "flags" is 0, encountering a malformation causes zero to be returned and *retlen is set so that ("s"?+?*retlen) is the next possible position in "s" that could begin a non-malformed character. Also, if UTF-8 warnings haven't been lexically disabled, a warning is raised. Some UTF-8 input sequences may contain multiple malformations. This function tries to find every possible one in each call, so multiple warnings can be raised for the same sequence.

Various ALLOW flags can be set in "flags" to allow (and not warn on) individual types of malformations, such as the sequence being overlong (that is, when there is a shorter sequence that can express the same code point; overlong sequences are expressly forbidden in the UTF-8 standard due to potential security issues). Another malformation example is the first byte of a character not being a legal first byte.

See utf8.h for the list of such flags. Even if allowed, this function generally returns the Unicode REPLACEMENT CHARACTER when it encounters a malformation. There are flags in utf8.h to override this behavior for the overlong malformations, but don't do that except for very specialized purposes.

The "UTF8_CHECK_ONLY" flag overrides the behavior when a non-allowed (by other flags) malformation is found. If this flag is set, the routine assumes that the caller will raise a warning, and this function will silently just set "retlen" to "-1" (cast to

"STRLEN") and return zero.

Note that this API requires disambiguation between successful decoding a "NUL" character, and an error return (unless the "UTF8_CHECK_ONLY" flag is set), as in both cases, 0 is returned, and, depending on the malformation, "retlen" may be set to 1. To disambiguate, upon a zero return, see if the first byte of "s" is 0 as well. If so, the input was a "NUL"; if not, the input had an error. Or you can use "utf8n_to_uvchr_error".

Certain code points are considered problematic. These are Unicode surrogates, Unicode non-characters, and code points above the Unicode maximum of 0x10FFFF. By default these are considered regular code points, but certain situations warrant special handling for them, which can be specified using the "flags" parameter. If "flags" contains "UTF8_DISALLOW_ILLEGAL_INTERCHANGE", all three classes are treated as malformations and handled as such. The flags "UTF8_DISALLOW_SURROGATE", "UTF8_DISALLOW_NONCHAR", and "UTF8_DISALLOW_SUPER" (meaning above the legal Unicode maximum) can be set to disallow these categories individually.

"UTF8_DISALLOW_ILLEGAL_INTERCHANGE" restricts the allowed inputs to the strict UTF-8 traditionally defined by Unicode. Use "UTF8_DISALLOW_ILLEGAL_C9_INTERCHANGE" to use the strictness definition given by Unicode Corrigendum #9 <https://www.unicode.org/versions/corrigendum9.html>. The difference between traditional strictness and C9 strictness is that the latter does not forbid non-character code points. (They are still discouraged, however.) For more discussion see "Noncharacter code points" in perlunicode.

The flags "UTF8_WARN_ILLEGAL_INTERCHANGE", "UTF8_WARN_ILLEGAL_C9_INTERCHANGE", "UTF8_WARN_SURROGATE", "UTF8_WARN_NONCHAR", and "UTF8_WARN_SUPER" will cause warning messages to be raised for their respective categories, but otherwise the code points are considered valid (not malformations). To get a category to both be treated as a malformation and raise a warning, specify both the WARN and DISALLOW flags. (But note that warnings are not raised if lexically disabled nor if "UTF8_CHECK_ONLY" is also specified.)

Extremely high code points were never specified in any standard, and require an extension to UTF-8 to express, which Perl does. It is likely that programs written in something other than Perl would not be able to read files that contain these; nor would Perl understand files written by something that uses a different extension. For

these reasons, there is a separate set of flags that can warn and/or disallow these extremely high code points, even if other above-Unicode ones are accepted. They are the "UTF8_WARN_PERL_EXTENDED" and "UTF8_DISALLOW_PERL_EXTENDED" flags. For more information see "UTF8_GOT_PERL_EXTENDED". Of course "UTF8_DISALLOW_SUPER" will treat all above-Unicode code points, including these, as malformations. (Note that the Unicode standard considers anything above 0x10FFFF to be illegal, but there are standards predating it that allow up to 0x7FFF_FFFF (2**31 -1))

A somewhat misleadingly named synonym for "UTF8_WARN_PERL_EXTENDED" is retained for backward compatibility: "UTF8_WARN_ABOVE_31_BIT". Similarly, "UTF8_DISALLOW_ABOVE_31_BIT" is usable instead of the more accurately named "UTF8_DISALLOW_PERL_EXTENDED". The names are misleading because these flags can apply to code points that actually do fit in 31 bits. This happens on EBCDIC platforms, and sometimes when the overlong malformation is also present. The new names accurately describe the situation in all cases.

All other code points corresponding to Unicode characters, including private use and those yet to be assigned, are never considered malformed and never warn.

UV utf8n_to_uvchr(const U8 *s, STRLEN curlen, STRLEN *retlen,
const U32 flags)

"utf8n_to_uvchr_error"

THIS FUNCTION SHOULD BE USED IN ONLY VERY SPECIALIZED CIRCUMSTANCES. Most code should use "utf8_to_uvchr_buf"() rather than call this directly.

This function is for code that needs to know what the precise malformation(s) are when an error is found. If you also need to know the generated warning messages, use "utf8n_to_uvchr_msgs"() instead.

It is like "utf8n_to_uvchr" but it takes an extra parameter placed after all the others, "errors". If this parameter is 0, this function behaves identically to "utf8n_to_uvchr". Otherwise, "errors" should be a pointer to a "U32" variable, which this function sets to indicate any errors found. Upon return, if *errors is 0, there were no errors found. Otherwise, *errors is the bit-wise "OR" of the bits described in the list below. Some of these bits will be set if a malformation is found, even if the input "flags" parameter indicates that the given malformation is allowed; those exceptions are noted:

"UTF8_GOT_PERL_EXTENDED"

The input sequence is not standard UTF-8, but a Perl extension. This bit is set only if the input "flags" parameter contains either the "UTF8_DISALLOW_PERL_EXTENDED" or the "UTF8_WARN_PERL_EXTENDED" flags.

Code points above 0x7FFF_FFFF (2**31 - 1) were never specified in any standard, and so some extension must be used to express them. Perl uses a natural extension to UTF-8 to represent the ones up to 2**36-1, and invented a further extension to represent even higher ones, so that any code point that fits in a 64-bit word can be represented. Text using these extensions is not likely to be portable to non-Perl code. We lump both of these extensions together and refer to them as Perl extended UTF-8. There exist other extensions that people have invented, incompatible with Perl's.

On EBCDIC platforms starting in Perl v5.24, the Perl extension for representing extremely high code points kicks in at 0x3FFF_FFFF (2**30 -1), which is lower than on ASCII. Prior to that, code points 2**31 and higher were simply unrepresentable, and a different, incompatible method was used to represent code points between 2**30 and 2**31 - 1.

On both platforms, ASCII and EBCDIC, "UTF8_GOT_PERL_EXTENDED" is set if Perl extended UTF-8 is used.

In earlier Perls, this bit was named "UTF8_GOT_ABOVE_31_BIT", which you still may use for backward compatibility. That name is misleading, as this flag may be set when the code point actually does fit in 31 bits. This happens on EBCDIC platforms, and sometimes when the overlong malformation is also present. The new name accurately describes the situation in all cases.

"UTF8_GOT_CONTINUATION"

The input sequence was malformed in that the first byte was a UTF-8 continuation byte.

"UTF8_GOT_EMPTY"

The input "curlen" parameter was 0.

"UTF8_GOT_LONG"

The input sequence was malformed in that there is some other sequence that evaluates to the same code point, but that sequence is shorter than this one.

Until Unicode 3.1, it was legal for programs to accept this malformation, but it was discovered that this created security issues.

"UTF8_GOT_NONCHAR"

The code point represented by the input UTF-8 sequence is for a Unicode non-character code point. This bit is set only if the input "flags" parameter contains either the "UTF8_DISALLOW_NONCHAR" or the "UTF8_WARN_NONCHAR" flags.

"UTF8_GOT_NON_CONTINUATION"

The input sequence was malformed in that a non-continuation type byte was found in a position where only a continuation type one should be. See also "UTF8_GOT_SHORT".

"UTF8_GOT_OVERFLOW"

The input sequence was malformed in that it is for a code point that is not representable in the number of bits available in an IV on the current platform.

"UTF8_GOT_SHORT"

The input sequence was malformed in that "curlen" is smaller than required for a complete sequence. In other words, the input is for a partial character sequence. "UTF8_GOT_SHORT" and "UTF8_GOT_NON_CONTINUATION" both indicate a too short sequence. The difference is that "UTF8_GOT_NON_CONTINUATION" indicates always that there is an error, while "UTF8_GOT_SHORT" means that an incomplete sequence was looked at. If no other flags are present, it means that the sequence was valid as far as it went. Depending on the application, this could mean one of three things:

- The "curlen" length parameter passed in was too small, and the function was prevented from examining all the necessary bytes.
- The buffer being looked at is based on reading data, and the data received so far stopped in the middle of a character, so that the next read will read the remainder of this character. (It is up to the caller to deal with the split bytes somehow.)
- This is a real error, and the partial sequence is all we're going to get.

"UTF8_GOT_SUPER"

The input sequence was malformed in that it is for a non-Unicode code point; that is, one above the legal Unicode maximum. This bit is set only if the input "flags" parameter contains either the "UTF8_DISALLOW_SUPER" or the "UTF8_WARN_SUPER" flags.

"UTF8_GOT_SURROGATE"

The input sequence was malformed in that it is for a -Unicode UTF-16 surrogate code point. This bit is set only if the input "flags" parameter contains either the "UTF8_DISALLOW_SURROGATE" or the "UTF8_WARN_SURROGATE" flags.

To do your own error handling, call this function with the "UTF8_CHECK_ONLY" flag to suppress any warnings, and then examine the *errors return.

```
 UV  utf8n_to_uvchr_error(const U8 *s, STRLEN curlen,
                          STRLEN *retlen, const U32 flags,
                          U32 * errors)
```

"utf8n_to_uvchr_msgs"

THIS FUNCTION SHOULD BE USED IN ONLY VERY SPECIALIZED CIRCUMSTANCES. Most code should use "utf8_to_uvchr_buf"() rather than call this directly.

This function is for code that needs to know what the precise malformation(s) are when an error is found, and wants the corresponding warning and/or error messages to be returned to the caller rather than be displayed. All messages that would have been displayed if all lexical warnings are enabled will be returned.

It is just like "utf8n_to_uvchr_error" but it takes an extra parameter placed after all the others, "msgs". If this parameter is 0, this function behaves identically to "utf8n_to_uvchr_error". Otherwise, "msgs" should be a pointer to an "AV *" variable, in which this function creates a new AV to contain any appropriate messages. The elements of the array are ordered so that the first message that would have been displayed is in the 0th element, and so on. Each element is a hash with three key-value pairs, as follows:

"text"

The text of the message as a "SVpv".

"warn_categories"

The warning category (or categories) packed into a "SVuv".

"flag"

A single flag bit associated with this message, in a "SVuv". The bit corresponds to some bit in the *errors return value, such as "UTF8_GOT_LONG".

It's important to note that specifying this parameter as non-null will cause any warnings this function would otherwise generate to be suppressed, and instead be placed in *msgs. The caller can check the lexical warnings state (or not) when choosing what to do with the returned messages.

If the flag "UTF8_CHECK_ONLY" is passed, no warnings are generated, and hence no AV is created.

The caller, of course, is responsible for freeing any returned AV.

```
 UV  utf8n_to_uvchr_msgs(const U8 *s, STRLEN curlen,
                         STRLEN *retlen, const U32 flags,
                         U32 * errors, AV ** msgs)
```

"UTF8SKIP"

returns the number of bytes a non-malformed UTF-8 encoded character whose first (perhaps only) byte is pointed to by "s".

If there is a possibility of malformed input, use instead:

"UTF8_SAFE_SKIP" if you know the maximum ending pointer in the buffer pointed to by "s"; or

"UTF8_CHK_SKIP" if you don't know it.

It is better to restructure your code so the end pointer is passed down so that you know what it actually is at the point of this call, but if that isn't possible, "UTF8_CHK_SKIP" can minimize the chance of accessing beyond the end of the input buffer.

```
 STRLEN  UTF8SKIP(char* s)
```

"UTF8_CHK_SKIP"

This is a safer version of "UTF8SKIP", but still not as safe as "UTF8_SAFE_SKIP".

This version doesn't blindly assume that the input string pointed to by "s" is well-formed, but verifies that there isn't a NUL terminating character before the expected end of the next character in "s". The length "UTF8_CHK_SKIP" returns stops just before any such NUL.

Perl tends to add NULs, as an insurance policy, after the end of strings in SV's, so it is likely that using this macro will prevent inadvertent reading beyond the end of the input buffer, even if it is malformed UTF-8.

This macro is intended to be used by XS modules where the inputs could be malformed, and it isn't feasible to restructure to use the safer "UTF8_SAFE_SKIP", for example when interfacing with a C library.

```
 STRLEN  UTF8_CHK_SKIP(char* s)
```

"utf8_distance"

Returns the number of UTF-8 characters between the UTF-8 pointers "a" and "b".

WARNING: use only if you *know* that the pointers point inside the same UTF-8 buffer.

IV utf8_distance(const U8 *a, const U8 *b)

"utf8_hop"

Return the UTF-8 pointer "s" displaced by "off" characters, either forward or backward.

WARNING: do not use the following unless you *know* "off" is within the UTF-8 data pointed to by "s" *and* that on entry "s" is aligned on the first byte of character or just after the last byte of a character.

U8* utf8_hop(const U8 *s, SSize_t off)

"utf8_hop_back"

Return the UTF-8 pointer "s" displaced by up to "off" characters, backward.

"off" must be non-positive.

"s" must be after or equal to "start".

When moving backward it will not move before "start".

Will not exceed this limit even if the string is not valid "UTF-8".

U8* utf8_hop_back(const U8 *s, SSize_t off, const U8 *start)

"utf8_hop_forward"

Return the UTF-8 pointer "s" displaced by up to "off" characters, forward.

"off" must be non-negative.

"s" must be before or equal to "end".

When moving forward it will not move beyond "end".

Will not exceed this limit even if the string is not valid "UTF-8".

U8* utf8_hop_forward(const U8 *s, SSize_t off, const U8 *end)

"utf8_hop_safe"

Return the UTF-8 pointer "s" displaced by up to "off" characters, either forward or backward.

When moving backward it will not move before "start".

When moving forward it will not move beyond "end".

Will not exceed those limits even if the string is not valid "UTF-8".

U8* utf8_hop_safe(const U8 *s, SSize_t off, const U8 *start,
                   const U8 *end)

"UTF8_IS_INVARIANT"

Evaluates to 1 if the byte "c" represents the same character when encoded in UTF-8 as

when not; otherwise evaluates to 0. UTF-8 invariant characters can be copied as-is when converting to/from UTF-8, saving time.

In spite of the name, this macro gives the correct result if the input string from which "c" comes is not encoded in UTF-8.

See "UVCHR_IS_INVARIANT" for checking if a UV is invariant.

```
bool  UTF8_IS_INVARIANT(char c)
```

"UTF8_IS_NONCHAR"

Evaluates to non-zero if the first few bytes of the string starting at "s" and looking no further than "e?-?1" are well-formed UTF-8 that represents one of the Unicode non-character code points; otherwise it evaluates to 0. If non-zero, the value gives how many bytes starting at "s" comprise the code point's representation.

```
bool  UTF8_IS_NONCHAR(const U8 *s, const U8 *e)
```

"UTF8_IS_SUPER"

Recall that Perl recognizes an extension to UTF-8 that can encode code points larger than the ones defined by Unicode, which are 0..0x10FFFF.

This macro evaluates to non-zero if the first few bytes of the string starting at "s" and looking no further than "e?-?1" are from this UTF-8 extension; otherwise it evaluates to 0. If non-zero, the value gives how many bytes starting at "s" comprise the code point's representation.

0 is returned if the bytes are not well-formed extended UTF-8, or if they represent a code point that cannot fit in a UV on the current platform. Hence this macro can give different results when run on a 64-bit word machine than on one with a 32-bit word size.

Note that it is illegal to have code points that are larger than what can fit in an IV on the current machine.

```
bool  UTF8_IS_SUPER(const U8 *s, const U8 *e)
```

"UTF8_IS_SURROGATE"

Evaluates to non-zero if the first few bytes of the string starting at "s" and looking no further than "e?-?1" are well-formed UTF-8 that represents one of the Unicode surrogate code points; otherwise it evaluates to 0. If non-zero, the value gives how many bytes starting at "s" comprise the code point's representation.

```
bool  UTF8_IS_SURROGATE(const U8 *s, const U8 *e)
```

"utf8_length"

Returns the number of characters in the sequence of UTF-8-encoded bytes starting at "s" and ending at the byte just before "e". If <s> and <e> point to the same place, it returns 0 with no warning raised.

If "e < s" or if the scan would end up past "e", it raises a UTF8 warning and returns the number of valid characters.

STRLEN utf8_length(const U8* s, const U8 *e)

"UTF8_MAXBYTES"

The maximum width of a single UTF-8 encoded character, in bytes.

NOTE: Strictly speaking Perl's UTF-8 should not be called UTF-8 since UTF-8 is an encoding of Unicode, and Unicode's upper limit, 0x10FFFF, can be expressed with 4 bytes. However, Perl thinks of UTF-8 as a way to encode non-negative integers in a binary format, even those above Unicode.

"UTF8_MAXBYTES_CASE"

The maximum number of UTF-8 bytes a single Unicode character can uppercase/lowercase/titlecase/fold into.

"UTF8_SAFE_SKIP"

returns 0 if "s?>=?e"; otherwise returns the number of bytes in the UTF-8 encoded character whose first byte is pointed to by "s". But it never returns beyond "e".

On DEBUGGING builds, it asserts that "s?<=?e".

STRLEN UTF8_SAFE_SKIP(char* s, char* e)

"UTF8_SKIP"

This is a synonym for "UTF8SKIP"

STRLEN UTF8_SKIP(char* s)

"utf8_to_bytes"

NOTE: "utf8_to_bytes" is experimental and may change or be removed without notice.

Converts a string "s" of length *lenp from UTF-8 into native byte encoding. Unlike "bytes_to_utf8", this over-writes the original string, and updates *lenp to contain the new length. Returns zero on failure (leaving "s" unchanged) setting *lenp to -1.

Upon successful return, the number of variants in the string can be computed by having saved the value of *lenp before the call, and subtracting the after-call value of *lenp from it.

If you need a copy of the string, see "bytes_from_utf8".

U8* utf8_to_bytes(U8 *s, STRLEN *lenp)

"utf8_to_uvchr"

"DEPRECATED!" It is planned to remove "utf8_to_uvchr" from a future release of Perl. Do not use it for new code; remove it from existing code.

Returns the native code point of the first character in the string "s" which is assumed to be in UTF-8 encoding; "retlen" will be set to the length, in bytes, of that character.

Some, but not all, UTF-8 malformations are detected, and in fact, some malformed input could cause reading beyond the end of the input buffer, which is why this function is deprecated. Use "utf8_to_uvchr_buf" instead.

If "s" points to one of the detected malformations, and UTF8 warnings are enabled, zero is returned and *retlen is set (if "retlen" isn't "NULL") to -1. If those warnings are off, the computed value if well-defined (or the Unicode REPLACEMENT CHARACTER, if not) is silently returned, and *retlen is set (if "retlen" isn't NULL) so that ("s"?+?*retlen) is the next possible position in "s" that could begin a non-malformed character. See "utf8n_to_uvchr" for details on when the REPLACEMENT CHARACTER is returned.

UV utf8_to_uvchr(const U8 *s, STRLEN *retlen)

"utf8_to_uvchr_buf"

Returns the native code point of the first character in the string "s" which is assumed to be in UTF-8 encoding; "send" points to 1 beyond the end of "s". *retlen will be set to the length, in bytes, of that character.

If "s" does not point to a well-formed UTF-8 character and UTF8 warnings are enabled, zero is returned and *retlen is set (if "retlen" isn't "NULL") to -1. If those warnings are off, the computed value, if well-defined (or the Unicode REPLACEMENT CHARACTER if not), is silently returned, and *retlen is set (if "retlen" isn't "NULL") so that ("s"?+?*retlen) is the next possible position in "s" that could begin a non-malformed character. See "utf8n_to_uvchr" for details on when the REPLACEMENT CHARACTER is returned.

UV utf8_to_uvchr_buf(const U8 *s, const U8 *send, STRLEN *retlen)

"UVCHR_IS_INVARIANT"

Evaluates to 1 if the representation of code point "cp" is the same whether or not it is encoded in UTF-8; otherwise evaluates to 0. UTF-8 invariant characters can be copied as-is when converting to/from UTF-8, saving time. "cp" is Unicode if above

255; otherwise is platform-native.

bool UVCHR_IS_INVARIANT(UV cp)

"UVCHR_SKIP"

returns the number of bytes required to represent the code point "cp" when encoded as UTF-8. "cp" is a native (ASCII or EBCDIC) code point if less than 255; a Unicode code point otherwise.

STRLEN UVCHR_SKIP(UV cp)

"uvchr_to_utf8"

Adds the UTF-8 representation of the native code point "uv" to the end of the string "d"; "d" should have at least "UVCHR_SKIP(uv)+1" (up to "UTF8_MAXBYTES+1") free bytes available. The return value is the pointer to the byte after the end of the new character. In other words,

```
d = uvchr_to_utf8(d, uv);
```

is the recommended wide native character-aware way of saying

```
*(d++) = uv;
```

This function accepts any code point from 0.."IV_MAX" as input. "IV_MAX" is typically 0x7FFF_FFFF in a 32-bit word.

It is possible to forbid or warn on non-Unicode code points, or those that may be problematic by using "uvchr_to_utf8_flags".

U8* uvchr_to_utf8(U8 *d, UV uv)

"uvchr_to_utf8_flags"

Adds the UTF-8 representation of the native code point "uv" to the end of the string "d"; "d" should have at least "UVCHR_SKIP(uv)+1" (up to "UTF8_MAXBYTES+1") free bytes available. The return value is the pointer to the byte after the end of the new character. In other words,

```
d = uvchr_to_utf8_flags(d, uv, flags);
```

or, in most cases,

```
d = uvchr_to_utf8_flags(d, uv, 0);
```

This is the Unicode-aware way of saying

```
*(d++) = uv;
```

If "flags" is 0, this function accepts any code point from 0.."IV_MAX" as input. "IV_MAX" is typically 0x7FFF_FFFF in a 32-bit word.

Specifying "flags" can further restrict what is allowed and not warned on, as follows:

If "uv" is a Unicode surrogate code point and "UNICODE_WARN_SURROGATE" is set, the function will raise a warning, provided UTF8 warnings are enabled. If instead "UNICODE_DISALLOW_SURROGATE" is set, the function will fail and return NULL. If both flags are set, the function will both warn and return NULL.

Similarly, the "UNICODE_WARN_NONCHAR" and "UNICODE_DISALLOW_NONCHAR" flags affect how the function handles a Unicode non-character.

And likewise, the "UNICODE_WARN_SUPER" and "UNICODE_DISALLOW_SUPER" flags affect the handling of code points that are above the Unicode maximum of 0x10FFFF. Languages other than Perl may not be able to accept files that contain these.

The flag "UNICODE_WARN_ILLEGAL_INTERCHANGE" selects all three of the above WARN flags; and "UNICODE_DISALLOW_ILLEGAL_INTERCHANGE" selects all three DISALLOW flags. "UNICODE_DISALLOW_ILLEGAL_INTERCHANGE" restricts the allowed inputs to the strict UTF-8 traditionally defined by Unicode. Similarly, "UNICODE_WARN_ILLEGAL_C9_INTERCHANGE" and "UNICODE_DISALLOW_ILLEGAL_C9_INTERCHANGE" are shortcuts to select the above-Unicode and surrogate flags, but not the non-character ones, as defined in Unicode Corrigendum #9 <https://www.unicode.org/versions/corrigendum9.html>. See "Noncharacter code points" in perlunicode.

Extremely high code points were never specified in any standard, and require an extension to UTF-8 to express, which Perl does. It is likely that programs written in something other than Perl would not be able to read files that contain these; nor would Perl understand files written by something that uses a different extension. For these reasons, there is a separate set of flags that can warn and/or disallow these extremely high code points, even if other above-Unicode ones are accepted. They are the "UNICODE_WARN_PERL_EXTENDED" and "UNICODE_DISALLOW_PERL_EXTENDED" flags. For more information see "UTF8_GOT_PERL_EXTENDED". Of course "UNICODE_DISALLOW_SUPER" will treat all above-Unicode code points, including these, as malformations. (Note that the Unicode standard considers anything above 0x10FFFF to be illegal, but there are standards predating it that allow up to 0x7FFF_FFFF (2**31 -1))

A somewhat misleadingly named synonym for "UNICODE_WARN_PERL_EXTENDED" is retained for backward compatibility: "UNICODE_WARN_ABOVE_31_BIT". Similarly, "UNICODE_DISALLOW_ABOVE_31_BIT" is usable instead of the more accurately named "UNICODE_DISALLOW_PERL_EXTENDED". The names are misleading because on EBCDIC

platforms,these flags can apply to code points that actually do fit in 31 bits. The new names accurately describe the situation in all cases.

U8* uvchr_to_utf8_flags(U8 *d, UV uv, UV flags)

"uvchr_to_utf8_flags_msgs"

THIS FUNCTION SHOULD BE USED IN ONLY VERY SPECIALIZED CIRCUMSTANCES.

Most code should use ""uvchr_to_utf8_flags"()" rather than call this directly.

This function is for code that wants any warning and/or error messages to be returned to the caller rather than be displayed. All messages that would have been displayed if all lexical warnings are enabled will be returned.

It is just like "uvchr_to_utf8_flags" but it takes an extra parameter placed after all the others, "msgs". If this parameter is 0, this function behaves identically to "uvchr_to_utf8_flags". Otherwise, "msgs" should be a pointer to an "HV *" variable, in which this function creates a new HV to contain any appropriate messages. The hash has three key-value pairs, as follows:

"text"

The text of the message as a "SVpv".

"warn_categories"

The warning category (or categories) packed into a "SVuv".

"flag"

A single flag bit associated with this message, in a "SVuv". The bit corresponds to some bit in the *errors return value, such as "UNICODE_GOT_SURROGATE".

It's important to note that specifying this parameter as non-null will cause any warnings this function would otherwise generate to be suppressed, and instead be placed in *msgs. The caller can check the lexical warnings state (or not) when choosing what to do with the returned messages.

The caller, of course, is responsible for freeing any returned HV.

U8* uvchr_to_utf8_flags_msgs(U8 *d, UV uv, UV flags, HV ** msgs)

Utility Functions

"C_ARRAY_END"

Returns a pointer to one element past the final element of the input C array.

void * C_ARRAY_END(void *a)

"C_ARRAY_LENGTH"

Returns the number of elements in the input C array (so you want your zero-based

indices to be less than but not equal to).

STRLEN C_ARRAY_LENGTH(void *a)

"getcwd_sv"

Fill "sv" with current working directory

int getcwd_sv(SV* sv)

"IN_PERL_COMPILETIME"

Returns 1 if this macro is being called during the compilation phase of the program;

otherwise 0;

bool IN_PERL_COMPILETIME

"IN_PERL_RUNTIME"

Returns 1 if this macro is being called during the execution phase of the program;

otherwise 0;

bool IN_PERL_RUNTIME

"IS_SAFE_SYSCALL"

Same as "is_safe_syscall".

bool IS_SAFE_SYSCALL(NN const char *pv, STRLEN len,

NN const char *what, NN const char *op_name)

"is_safe_syscall"

Test that the given "pv" (with length "len") doesn't contain any internal "NUL"

characters. If it does, set "errno" to "ENOENT", optionally warn using the "syscalls"

category, and return FALSE.

Return TRUE if the name is safe.

"what" and "op_name" are used in any warning.

Used by the "IS_SAFE_SYSCALL()" macro.

bool is_safe_syscall(const char *pv, STRLEN len,

const char *what, const char *op_name)

"my_setenv"

A wrapper for the C library setenv(3). Don't use the latter, as the perl version has

desirable safeguards

void my_setenv(const char* nam, const char* val)

"Poison"

PoisonWith(0xEF) for catching access to freed memory.

void Poison(void* dest, int nitems, type)

"PoisonFree"

PoisonWith(0xEF) for catching access to freed memory.

void PoisonFree(void* dest, int nitems, type)

"PoisonNew"

PoisonWith(0xAB) for catching access to allocated but uninitialized memory.

void PoisonNew(void* dest, int nitems, type)

"PoisonWith"

Fill up memory with a byte pattern (a byte repeated over and over again) that hopefully catches attempts to access uninitialized memory.

void PoisonWith(void* dest, int nitems, type, U8 byte)

"StructCopy"

This is an architecture-independent macro to copy one structure to another.

void StructCopy(type *src, type *dest, type)

"sv_destroyable"

Dummy routine which reports that object can be destroyed when there is no sharing module present. It ignores its single SV argument, and returns 'true'. Exists to avoid test for a "NULL" function pointer and because it could potentially warn under some level of strict-ness.

bool sv_destroyable(SV *sv)

"sv_nosharing"

Dummy routine which "shares" an SV when there is no sharing module present. Or "locks" it. Or "unlocks" it. In other words, ignores its single SV argument. Exists to avoid test for a "NULL" function pointer and because it could potentially warn under some level of strict-ness.

void sv_nosharing(SV *sv)

Versioning

"new_version"

Returns a new version object based on the passed in SV:

```
SV *sv = new_version(SV *ver);
```

Does not alter the passed in ver SV. See "upg_version" if you want to upgrade the SV.

SV* new_version(SV *ver)

"PERL_REVISION"

"DEPRECATED!" It is planned to remove "PERL_REVISION" from a future release of Perl.

Do not use it for new code; remove it from existing code.

The major number component of the perl interpreter currently being compiled or executing. This has been 5 from 1993 into 2020.

Instead use one of the version comparison macros. See "PERL_VERSION_EQ".

"PERL_SUBVERSION"

"DEPRECATED!" It is planned to remove "PERL_SUBVERSION" from a future release of Perl. Do not use it for new code; remove it from existing code.

The micro number component of the perl interpreter currently being compiled or executing. In stable releases this gives the dot release number for maintenance updates. In development releases this gives a tag for a snapshot of the status at various points in the development cycle.

Instead use one of the version comparison macros. See "PERL_VERSION_EQ".

"PERL_VERSION"

"DEPRECATED!" It is planned to remove "PERL_VERSION" from a future release of Perl. Do not use it for new code; remove it from existing code.

The minor number component of the perl interpreter currently being compiled or executing. Between 1993 into 2020, this has ranged from 0 to 33.

Instead use one of the version comparison macros. See "PERL_VERSION_EQ".

"PERL_VERSION_EQ"

"PERL_VERSION_NE"

"PERL_VERSION_LT"

"PERL_VERSION_LE"

"PERL_VERSION_GT"

"PERL_VERSION_GE"

Returns whether or not the perl currently being compiled has the specified relationship to the perl given by the parameters. For example,

```
#if PERL_VERSION_GT(5,24,2)
  code that will only be compiled on perls after v5.24.2
#else
  fallback code
#endif
```

Note that this is usable in making compile-time decisions

You may use the special value '*' for the final number to mean ALL possible values for

it. Thus,

#if PERL_VERSION_EQ(5,31,'*')

means all perls in the 5.31 series. And

#if PERL_VERSION_NE(5,24,'*')

means all perls EXCEPT 5.24 ones. And

#if PERL_VERSION_LE(5,9,'*')

is effectively

#if PERL_VERSION_LT(5,10,0)

This means you don't have to think so much when converting from the existing deprecated "PERL_VERSION" to using this macro:

#if PERL_VERSION <= 9

becomes

#if PERL_VERSION_LE(5,9,'*')

```
bool  PERL_VERSION_EQ(const U8 major, const U8 minor,
                      const U8 patch)
```

"prescan_version"

Validate that a given string can be parsed as a version object, but doesn't actually perform the parsing. Can use either strict or lax validation rules. Can optionally set a number of hint variables to save the parsing code some time when tokenizing.

```
const char*  prescan_version(const char *s, bool strict,
                             const char** errstr, bool *sqv,
                             int *ssaw_decimal, int *swidth,
                             bool *salpha)
```

"scan_version"

Returns a pointer to the next character after the parsed version string, as well as upgrading the passed in SV to an RV.

Function must be called with an already existing SV like

```
sv = newSV(0);
s = scan_version(s, SV *sv, bool qv);
```

Performs some preprocessing to the string to ensure that it has the correct characteristics of a version. Flags the object if it contains an underscore (which denotes this is an alpha version). The boolean qv denotes that the version should be interpreted as if it had multiple decimals, even if it doesn't.

const char* scan_version(const char *s, SV *rv, bool qv)

"upg_version"

In-place upgrade of the supplied SV to a version object.

SV *sv = upg_version(SV *sv, bool qv);

Returns a pointer to the upgraded SV. Set the boolean qv if you want to force this SV to be interpreted as an "extended" version.

SV* upg_version(SV *ver, bool qv)

"vcmp"

Version object aware cmp. Both operands must already have been converted into version objects.

int vcmp(SV *lhv, SV *rhv)

"vnormal"

Accepts a version object and returns the normalized string representation. Call like:

sv = vnormal(rv);

NOTE: you can pass either the object directly or the SV contained within the RV.

The SV returned has a refcount of 1.

SV* vnormal(SV *vs)

"vnumify"

Accepts a version object and returns the normalized floating point representation.

Call like:

sv = vnumify(rv);

NOTE: you can pass either the object directly or the SV contained within the RV.

The SV returned has a refcount of 1.

SV* vnumify(SV *vs)

"vstringify"

In order to maintain maximum compatibility with earlier versions of Perl, this function will return either the floating point notation or the multiple dotted notation, depending on whether the original version contained 1 or more dots, respectively.

The SV returned has a refcount of 1.

SV* vstringify(SV *vs)

"vverify"

Validates that the SV contains valid internal structure for a version object. It may

be passed either the version object (RV) or the hash itself (HV). If the structure is valid, it returns the HV. If the structure is invalid, it returns NULL.

```
SV *hv = vverify(sv);
```

Note that it only confirms the bare minimum structure (so as not to get confused by derived classes which may contain additional hash entries):

? The SV is an HV or a reference to an HV

? The hash contains a "version" key

? The "version" key has a reference to an AV as its value

```
SV*  vverify(SV *vs)
```

Warning and Dieing

In all these calls, the "U32 wn" parameters are warning category constants. You can see the ones currently available in "Category Hierarchy" in warnings, just capitalize all letters in the names and prefix them by "WARN_". So, for example, the category "void" used in a perl program becomes "WARN_VOID" when used in XS code and passed to one of the calls below.

"ckWARN"

"ckWARN2"

"ckWARN3"

"ckWARN4"

These return a boolean as to whether or not warnings are enabled for any of the warning category(ies) parameters: "w", "w1", ...

Should any of the categories by default be enabled even if not within the scope of "use?warnings", instead use the "ckWARN_d" macros.

The categories must be completely independent, one may not be subclassed from the other.

```
bool  ckWARN (U32 w)
bool  ckWARN2(U32 w1, U32 w2)
bool  ckWARN3(U32 w1, U32 w2, U32 w3)
bool  ckWARN4(U32 w1, U32 w2, U32 w3, U32 w4)
```

"ckWARN_d"

"ckWARN2_d"

"ckWARN3_d"

"ckWARN4_d"

Like "ckWARN", but for use if and only if the warning category(ies) is by default enabled even if not within the scope of "use?warnings".

```
bool  ckWARN_d (U32 w)
bool  ckWARN2_d(U32 w1, U32 w2)
bool  ckWARN3_d(U32 w1, U32 w2, U32 w3)
bool  ckWARN4_d(U32 w1, U32 w2, U32 w3, U32 w4)
```

"ck_warner"

"ck_warner_d"

If none of the warning categories given by "err" are enabled, do nothing; otherwise call "warner" or "warner_nocontext" with the passed-in parameters;.

"err" must be one of the "packWARN", "packWARN2", "packWARN3", "packWARN4" macros populated with the appropriate number of warning categories.

The two forms differ only in that "ck_warner_d" should be used if warnings for any of the categories are by default enabled.

NOTE: "ck_warner" must be explicitly called as "Perl_ck_warner" with an "aTHX_" parameter.

NOTE: "ck_warner_d" must be explicitly called as "Perl_ck_warner_d" with an "aTHX_" parameter.

```
void  Perl_ck_warner(pTHX_ U32 err, const char* pat, ...)
```

"CLEAR_ERRSV"

Clear the contents of $@, setting it to the empty string.

This replaces any read-only SV with a fresh SV and removes any magic.

```
void  CLEAR_ERRSV()
```

"croak"

"croak_nocontext"

These are XS interfaces to Perl's "die" function.

They take a sprintf-style format pattern and argument list, which are used to generate a string message. If the message does not end with a newline, then it will be extended with some indication of the current location in the code, as described for "mess_sv".

The error message will be used as an exception, by default returning control to the nearest enclosing "eval", but subject to modification by a $SIG{__DIE__} handler. In any case, these croak functions never return normally.

For historical reasons, if "pat" is null then the contents of "ERRSV" ($@) will be used as an error message or object instead of building an error message from arguments. If you want to throw a non-string object, or build an error message in an SV yourself, it is preferable to use the "croak_sv" function, which does not involve clobbering "ERRSV".

The two forms differ only in that "croak_nocontext" does not take a thread context ("aTHX") parameter. It is usually preferred as it takes up fewer bytes of code than plain "Perl_croak", and time is rarely a critical resource when you are about to throw an exception.

NOTE: "croak" must be explicitly called as "Perl_croak" with an "aTHX_" parameter.

```
 void  Perl_croak     (pTHX_ const char* pat, ...)
 void  croak_nocontext(const char* pat, ...)
```

"croak_no_modify"

This encapsulates a common reason for dying, generating terser object code than using the generic "Perl_croak". It is exactly equivalent to "Perl_croak(aTHX_ "%s", PL_no_modify)" (which expands to something like "Modification of a read-only value attempted").

Less code used on exception code paths reduces CPU cache pressure.

```
 void  croak_no_modify()
```

"croak_sv"

This is an XS interface to Perl's "die" function.

"baseex" is the error message or object. If it is a reference, it will be used as-is. Otherwise it is used as a string, and if it does not end with a newline then it will be extended with some indication of the current location in the code, as described for "mess_sv".

The error message or object will be used as an exception, by default returning control to the nearest enclosing "eval", but subject to modification by a $SIG{__DIE__} handler. In any case, the "croak_sv" function never returns normally.

To die with a simple string message, the "croak" function may be more convenient.

```
 void  croak_sv(SV *baseex)
```

"die"

Behaves the same as "croak", except for the return type. It should be used only where the "OP *" return type is required. The function never actually returns.

NOTE: "die" must be explicitly called as "Perl_die" with an "aTHX_" parameter.

```
OP*  Perl_die(pTHX_ const char* pat, ...)
```

"die_sv"

"die_nocontext"

These ehave the same as "croak_sv", except for the return type. It should be used only where the "OP *" return type is required. The functions never actually return.

The two forms differ only in that "die_nocontext" does not take a thread context ("aTHX") parameter, so is used in situations where the caller doesn't already have the thread context.

```
OP*  die_sv       (SV *baseex)
OP*  die_nocontext(const char* pat, ...)
```

"ERRSV"

Returns the SV for $@, creating it if needed.

```
SV *  ERRSV
```

"packWARN"

"packWARN2"

"packWARN3"

"packWARN4"

These macros are used to pack warning categories into a single U32 to pass to macros and functions that take a warning category parameter. The number of categories to pack is given by the name, with a corresponding number of category parameters passed.

```
U32  packWARN (U32 w1)
U32  packWARN2(U32 w1, U32 w2)
U32  packWARN3(U32 w1, U32 w2, U32 w3)
U32  packWARN4(U32 w1, U32 w2, U32 w3, U32 w4)
```

"PL_curcop"

The currently active COP (control op) roughly representing the current statement in the source.

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

```
COP*  PL_curcop
```

"PL_curstash"

The stash for the package code will be compiled into.

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

HV* PL_curstash

"PL_defgv"

The GV representing *_. Useful for access to $_.

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

GV * PL_defgv

"SANE_ERRSV"

Clean up ERRSV so we can safely set it.

This replaces any read-only SV with a fresh writable copy and removes any magic.

void SANE_ERRSV()

"vcroak"

This is an XS interface to Perl's "die" function.

"pat" and "args" are a sprintf-style format pattern and encapsulated argument list. These are used to generate a string message. If the message does not end with a newline, then it will be extended with some indication of the current location in the code, as described for "mess_sv".

The error message will be used as an exception, by default returning control to the nearest enclosing "eval", but subject to modification by a $SIG{__DIE__} handler. In any case, the "croak" function never returns normally.

For historical reasons, if "pat" is null then the contents of "ERRSV" ($@) will be used as an error message or object instead of building an error message from arguments. If you want to throw a non-string object, or build an error message in an SV yourself, it is preferable to use the "croak_sv" function, which does not involve clobbering "ERRSV".

void vcroak(const char* pat, va_list* args)

"vwarn"

This is an XS interface to Perl's "warn" function.

This is like "warn", but "args" are an encapsulated argument list.

Unlike with "vcroak", "pat" is not permitted to be null.

void vwarn(const char* pat, va_list* args)

"vwarner"

This is like "warner", but "args" are an encapsulated argument list.

```
 void  vwarner(U32 err, const char* pat, va_list* args)
```

"warn"

"warn_nocontext"

These are XS interfaces to Perl's "warn" function.

They take a sprintf-style format pattern and argument list, which are used to generate a string message. If the message does not end with a newline, then it will be extended with some indication of the current location in the code, as described for "mess_sv".

The error message or object will by default be written to standard error, but this is subject to modification by a $SIG{__WARN__} handler.

Unlike with "croak", "pat" is not permitted to be null.

The two forms differ only in that "warn_nocontext" does not take a thread context ("aTHX") parameter, so is used in situations where the caller doesn't already have the thread context.

NOTE: "warn" must be explicitly called as "Perl_warn" with an "aTHX_" parameter.

```
 void  Perl_warn     (pTHX_ const char* pat, ...)
 void  warn_nocontext(const char* pat, ...)
```

"warner"

"warner_nocontext"

These output a warning of the specified category (or categories) given by "err", using the sprintf-style format pattern "pat", and argument list.

"err" must be one of the "packWARN", "packWARN2", "packWARN3", "packWARN4" macros populated with the appropriate number of warning categories. If any of the warning categories they specify is fatal, a fatal exception is thrown.

In any event a message is generated by the pattern and arguments. If the message does not end with a newline, then it will be extended with some indication of the current location in the code, as described for "mess_sv".

The error message or object will by default be written to standard error, but this is subject to modification by a $SIG{__WARN__} handler.

"pat" is not permitted to be null.

The two forms differ only in that "warner_nocontext" does not take a thread context ("aTHX") parameter, so is used in situations where the caller doesn't already have the

thread context.

These functions differ from the similarly named "warn" functions, in that the latter are for XS code to unconditionally display a warning, whereas these are for code that may be compiling a perl program, and does extra checking to see if the warning should be fatal.

NOTE: "warner" must be explicitly called as "Perl_warner" with an "aTHX_" parameter.

```
void  Perl_warner     (pTHX_ U32 err, const char* pat, ...)
void  warner_nocontext(U32 err, const char* pat, ...)
```

"warn_sv"

This is an XS interface to Perl's "warn" function.

"baseex" is the error message or object. If it is a reference, it will be used as-is. Otherwise it is used as a string, and if it does not end with a newline then it will be extended with some indication of the current location in the code, as described for "mess_sv".

The error message or object will by default be written to standard error, but this is subject to modification by a $SIG{__WARN__} handler.

To warn with a simple string message, the "warn" function may be more convenient.

```
void  warn_sv(SV *baseex)
```

XS

xsubpp compiles XS code into C. See "xsubpp" in perlutil.

"ax"

Variable which is setup by "xsubpp" to indicate the stack base offset, used by the "ST", "XSprePUSH" and "XSRETURN" macros. The "dMARK" macro must be called prior to setup the "MARK" variable.

```
I32  ax
```

"CLASS"

Variable which is setup by "xsubpp" to indicate the class name for a C++ XS constructor. This is always a "char*". See "THIS".

```
char*  CLASS
```

"dAX"

Sets up the "ax" variable. This is usually handled automatically by "xsubpp" by calling "dXSARGS".

```
dAX;
```

"dAXMARK"

Sets up the "ax" variable and stack marker variable "mark". This is usually handled automatically by "xsubpp" by calling "dXSARGS".

```
dAXMARK;
```

"dITEMS"

Sets up the "items" variable. This is usually handled automatically by "xsubpp" by calling "dXSARGS".

```
dITEMS;
```

"dMY_CXT_SV"

Now a placeholder that declares nothing

```
dMY_CXT_SV;
```

"dUNDERBAR"

Sets up any variable needed by the "UNDERBAR" macro. It used to define "padoff_du", but it is currently a noop. However, it is strongly advised to still use it for ensuring past and future compatibility.

```
dUNDERBAR;
```

"dXSARGS"

Sets up stack and mark pointers for an XSUB, calling "dSP" and "dMARK". Sets up the "ax" and "items" variables by calling "dAX" and "dITEMS". This is usually handled automatically by "xsubpp".

```
dXSARGS;
```

"dXSI32"

Sets up the "ix" variable for an XSUB which has aliases. This is usually handled automatically by "xsubpp".

```
dXSI32;
```

"items"

Variable which is setup by "xsubpp" to indicate the number of items on the stack. See "Variable-length Parameter Lists" in perlxs.

```
I32  items
```

"ix"

Variable which is setup by "xsubpp" to indicate which of an XSUB's aliases was used to invoke it. See "The ALIAS: Keyword" in perlxs.

```
I32  ix
```

"RETVAL"

Variable which is setup by "xsubpp" to hold the return value for an XSUB. This is always the proper type for the XSUB. See "The RETVAL Variable" in perlxs.

```
type  RETVAL
```

"ST"

Used to access elements on the XSUB's stack.

```
SV*  ST(int ix)
```

"THIS"

Variable which is setup by "xsubpp" to designate the object in a C++ XSUB. This is always the proper type for the C++ object. See "CLASS" and "Using XS With C++" in perlxs.

```
type  THIS
```

"UNDERBAR"

The SV* corresponding to the $_ variable. Works even if there is a lexical $_ in scope.

"XS"

Macro to declare an XSUB and its C parameter list. This is handled by "xsubpp". It is the same as using the more explicit "XS_EXTERNAL" macro; the latter is preferred.

"XS_EXTERNAL"

Macro to declare an XSUB and its C parameter list explicitly exporting the symbols.

"XS_INTERNAL"

Macro to declare an XSUB and its C parameter list without exporting the symbols. This is handled by "xsubpp" and generally preferable over exporting the XSUB symbols unnecessarily.

"XSPROTO"

Macro used by "XS_INTERNAL" and "XS_EXTERNAL" to declare a function prototype. You probably shouldn't be using this directly yourself.

Undocumented elements

The following functions have been flagged as part of the public API, but are currently undocumented. Use them at your own risk, as the interfaces are subject to change.

Functions that are not listed in this document are not intended for public use, and should NOT be used under any circumstances.

If you feel you need to use one of these functions, first send email to

perl5-porters@perl.org <mailto:perl5-porters@perl.org>. It may be that there is a good reason for the function not being documented, and it should be removed from this list; or it may just be that no one has gotten around to documenting it. In the latter case, you will be asked to submit a patch to document the function. Once your patch is accepted, it will indicate that the interface is stable (unless it is explicitly marked otherwise) and usable by you.

amagic_call gv_name_set PerlIO_context_layers

amagic_deref_call gv_SVadd PerlIO_fill

any_dup he_dup PerlIO_unread

atfork_lock hek_dup pmop_dump

atfork_unlock hv_delayfree_ent pop_scope

block_gimme hv_eiter_p pregfree

call_atexit hv_eiter_set ptr_table_fetch

call_list hv_free_ent ptr_table_free

clear_defarray hv_ksplit ptr_table_new

clone_params_del hv_name_set ptr_table_split

clone_params_new hv_placeholders_get ptr_table_store

CvDEPTH hv_placeholders_set push_scope

deb hv_rand_set re_compile

deb_nocontext hv_riter_p regdump

debop hv_riter_set repeatcpy

debprofdump init_stacks rsignal_state

debstack init_tm rvpv_dup

debstackptrs is_lvalue_sub save_adelete

dirp_dup leave_scope save_aelem

do_aspawn magic_dump save_aelem_flags

do_close markstack_grow save_alloc

do_join mfree save_generic_pvref

do_open mg_dup save_generic_svref

do_openn mg_size save_hdelete

doref mro_get_from_name save_helem

do_spawn mro_set_mro save_helem_flags

do_spawn_nowait my_chsize save_hints

do_sprintf my_cxt_init save_op

dounwind my_dirfd save_padsv_and_mortalize

dowantarray my_failure_exit save_pushi32ptr

dump_eval my_fflush_all save_pushptr

dump_form my_fork save_pushptrptr

dump_mstats my_pclose save_set_svflags

dump_sub my_popen save_shared_pvref

filter_del my_popen_list savestack_grow

fp_dup my_socketpair savestack_grow_cnt

get_context newANONATTRSUB save_vptr

get_mstats newANONHASH scan_vstring

get_op_descs newANONLIST seed

get_op_names newANONSUB set_context

get_ppaddr newAVREF share_hek

get_vtbl newCVREF si_dup

gp_dup newFORM ss_dup

gp_free newGVgen start_subparse

gp_ref newGVgen_flags sv_2pvbyte_flags

gv_add_by_type newGVREF sv_2pvutf8_flags

Gv_AMupdate newHVhv SvAMAGIC_off

gv_autoload_pv newHVREF SvAMAGIC_on

gv_autoload_pvn newIO sv_dup

gv_autoload_sv newMYSUB sv_dup_inc

gv_AVadd newPROG sv_peek

gv_dump new_stackinfo sys_intern_clear

gv_efullname3 newSVREF sys_intern_dup

gv_efullname4 op_refcnt_lock sys_intern_init

gv_fullname3 op_refcnt_unlock taint_env

gv_fullname4 parser_dup taint_proper

gv_handler perl_alloc_using unsharepvn

gv_HVadd PERL_BUILD_DATE vdeb

gv_IOadd perl_clone_using

Until May 1997, this document was maintained by Jeff Okamoto <okamoto@corp.hp.com>. It is now maintained as part of Perl itself.

With lots of help and suggestions from Dean Roehrich, Malcolm Beattie, Andreas Koenig, Paul Hudson, Ilya Zakharevich, Paul Marquess, Neil Bowers, Matthew Green, Tim Bunce, Spider Boardman, Ulrich Pfeifer, Stephen McCamant, and Gurusamy Sarathy.

API Listing originally by Dean Roehrich <roehrich@cray.com>.

Updated to be autogenerated from comments in the source by Benjamin Stuhl.

SEE ALSO

config.h, perlapio, perlcall, perlclib, perlfilter, perlguts, perlintern, perlinterp, perliol, perlmroapi, perlreguts, perlxs