NAME

perlapi - autogenerated documentation for the perl public API

DESCRIPTION

This file contains the documentation of the perl public API generated by *embed.pl*, specifically a listing of functions, macros, flags, and variables that may be used by extension writers. At the end is a list of functions which have yet to be documented. The interfaces of those are subject to change without notice. Anything not listed here 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 somethimes 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 listing below is alphabetical, case insensitive.

Array Manipulation Functions

av_clear Frees the 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_create_and_push

NOTE: this function 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.

```
void av_create_and_push(AV **const avp,
                        SV *const val)
```

av_create_and_unshift_one

NOTE: this function 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.

```
SV** av_create_and_unshift_one(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. Perl equivalent: `my$elem=delete($myarray[$idx]);` for the non-`G_DISCARD` version and a void-context `delete($myarray[$idx]);` for the `G_DISCARD` version.

```
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. The `key` is the index to which the array should be extended.

```
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(1) for more information on how to use this function on tied arrays.

The rough perl equivalent is `$myarray[$idx]`.

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

AvFILL Same as `av_top_index()`. Deprecated, use `av_top_index()` instead.

```
int 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 highest index in the array, so to get the size of the array you need to use `av_len(av)+1`. This is unlike "sv_len", which returns what you would expect.

```
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, $elem;`.

```
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: `$myarray[$key] = $val;`.

See ''Understanding the Magic of Tied Hashes and Arrays'' in perlguts(1) for more information on how to use this function on tied arrays.

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

av_tindex
          Same as `av_top_index()`.

```
int av_tindex(AV* av)
```

av_top_index
          Returns the highest index in the array. The number of elements in the array is `av_top_index(av)+1`. Returns -1 if the array is empty.

The Perl equivalent for this is `$#myarray`.

(A slightly shorter form is `av_tindex`.)

```
SSize_t av_top_index(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. You must then use `av_store` to assign values to these new elements.

Perl equivalent: `unshift@myarray,((undef)x$n);`

```
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_ form of this function 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()
          ```

sortsv   Sort an array. Here is an example:

          ```
           sortsv(AvARRAY(av), av_top_index(av)+1, Perl_sv_cmp_locale);
          ```

          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
          Sort an array, with various options.

          ```
           void sortsv_flags(SV** array, size_t num_elts,
           SVCOMPARE_t cmp, U32 flags)
          ```

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_ form of this function 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_ form of this function 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_ form of this function 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_ form of this function is deprecated.

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

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

```
ENTER;
```

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

NOTE: the perl_ form of this function 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.

NOTE: the perl_ form of this function is deprecated.

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

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

```
FREETMPS;
```

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

```
LEAVE;
```

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

```
SAVETMPS;
```

Character case changing

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
    Converts the UTF-8 encoded character at `p` 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 just above, that there may be more.)

The input character at `p` is assumed to be well-formed.

```
UV toFOLD_utf8(U8* p, 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 just above, that there may be more.)

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

toLOWER

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

```
U8 toLOWER(U8 ch)
```

toLOWER_L1

Converts the specified Latin1 character to lowercase. The results are undefined if the input doesn't fit in a byte.

```
U8 toLOWER_L1(U8 ch)
```

toLOWER_LC

Converts the specified character to lowercase using the current locale's rules, if possible; otherwise returns the input character itself.

```
U8 toLOWER_LC(U8 ch)
```

toLOWER_utf8

Converts the UTF-8 encoded character at p to its lowercase 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 lowercase version may be longer than the original character.

The first code point of the lowercased version is returned (but note, as explained just above, that there may be more.)

The input character at p is assumed to be well-formed.

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

toLOWER_uvchr

Converts the code point cp to its lowercase 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 lowercase version may be longer than the original character.

The first code point of the lowercased version is returned (but note, as explained just above, that there may be more.)

```
UV toLOWER_uvchr(UV cp, 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

Converts the UTF-8 encoded character at p 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 just above, that there may be more.)

The input character at p is assumed to be well-formed.

```
                 UV toTITLE_utf8(U8* p, 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 just above, 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(U8 ch)
```

toUPPER_utf8

Converts the UTF-8 encoded character at `p` 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 just above, that there may be more.)

The input character at `p` is assumed to be well-formed.

```
                 UV toUPPER_utf8(U8* p, 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 just above, 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 an octet (either a `char` or a `U8`) as input and returns a boolean as to whether or not the character represented by that octet 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".

Variant `isFOO_L1` imposes the Latin-1 (or EBCDIC equivlalent) 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).

Variant `isFOO_uvchr` is like the `isFOO_L1` variant, but accepts any UV code point as input. 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.

Variant `isFOO_utf8` is like `isFOO_uvchr`, but the input is a pointer to a (known to be well-formed) UTF-8 encoded string (`U8*` or `char*`, and possibly containing embedded `NUL` characters). The classification of just the first (possibly multi-byte) character in the string is tested.

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` is like `isFOO_LC`, but is defined on any UV. It returns the same as `isFOO_LC` for input code points less than 256, and returns the hard-coded, not-affected-by-locale, Unicode results for larger ones.

Variant `isFOO_LC_utf8` is like `isFOO_LC_uvchr`, but the input is a pointer to a (known to be well-formed) UTF-8 encoded string (`U8*` or `char*`, and possibly containing embedded `NUL` characters). The classification of just the first (possibly multi-byte) character in the string is tested.

isALPHA

Returns a boolean indicating whether the specified character is an alphabetic character, analogous to `m/[[:alpha:]]/`. See the top of this section for an explanation of variants `isALPHA_A`, `isALPHA_L1`, `isALPHA_uvchr`, `isALPHA_utf8`, `isALPHA_LC`, `isALPHA_LC_uvchr`, and `isALPHA_LC_utf8`.

```
bool isALPHA(char ch)
```

isALPHANUMERIC

Returns a boolean indicating whether the specified character is a either an alphabetic character or decimal digit, analogous to `m/[[:alnum:]]/`. See the top of this section for an explanation of variants `isALPHANUMERIC_A`, `isALPHANUMERIC_L1`, `isALPHANUMERIC_uvchr`, `isALPHANUMERIC_utf8`, `isALPHANUMERIC_LC`, `isALPHANUMERIC_LC_uvchr`, and `isALPHANUMERIC_LC_utf8`.

```
bool isALPHANUMERIC(char ch)
```

isASCII   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 variants `isASCII_uvchr`, `isASCII_utf8`, `isASCII_LC`, `isASCII_LC_uvchr`, and `isASCII_LC_utf8`. 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` will work properly on any string encoded or not in UTF-8.

```
bool isASCII(char ch)
```

isBLANK

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 variants `isBLANK_A`, `isBLANK_L1`, `isBLANK_uvchr`, `isBLANK_utf8`, `isBLANK_LC`, `isBLANK_LC_uvchr`, and `isBLANK_LC_utf8`. 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(char ch)
```

isCNTRL

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 variants `isCNTRL_A`, `isCNTRL_L1`, `isCNTRL_uvchr`, `isCNTRL_utf8`, `isCNTRL_LC`, `isCNTRL_LC_uvchr`, and `isCNTRL_LC_utf8` On EBCDIC platforms, you almost always want to use the `isCNTRL_L1` variant.

```
bool isCNTRL(char ch)
```

isDIGIT 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 variants `isDIGIT_uvchr`, `isDIGIT_utf8`, `isDIGIT_LC`, `isDIGIT_LC_uvchr`, and `isDIGIT_LC_utf8`.

```
bool isDIGIT(char ch)
```

isGRAPH

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 variants `isGRAPH_A`, `isGRAPH_L1`, `isGRAPH_uvchr`, `isGRAPH_utf8`, `isGRAPH_LC`, `isGRAPH_LC_uvchr`, and `isGRAPH_LC_utf8`.

```
bool isGRAPH(char ch)
```

isIDCONT

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 variants `isIDCONT_A`, `isIDCONT_L1`, `isIDCONT_uvchr`, `isIDCONT_utf8`, `isIDCONT_LC`, `isIDCONT_LC_uvchr`, and `isIDCONT_LC_utf8`.

```
bool isIDCONT(char ch)
```

isIDFIRST

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 variants `isIDFIRST_A`, `isIDFIRST_L1`, `isIDFIRST_uvchr`, `isIDFIRST_utf8`, `isIDFIRST_LC`, `isIDFIRST_LC_uvchr`, and `isIDFIRST_LC_utf8`.

```
bool isIDFIRST(char ch)
```

isLOWER

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 variants `isLOWER_A`, `isLOWER_L1`, `isLOWER_uvchr`, `isLOWER_utf8`, `isLOWER_LC`, `isLOWER_LC_uvchr`, and `isLOWER_LC_utf8`.

```
bool isLOWER(char ch)
```

isOCTAL

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(char ch)
```

isPRINT

> 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 variants `isPRINT_A`, `isPRINT_L1`, `isPRINT_uvchr`, `isPRINT_utf8`, `isPRINT_LC`, `isPRINT_LC_uvchr`, and `isPRINT_LC_utf8`.
>
> ```
> bool isPRINT(char ch)
> ```

isPSXSPC

> (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 variants `isPSXSPC_A`, `isPSXSPC_L1`, `isPSXSPC_uvchr`, `isPSXSPC_utf8`, `isPSXSPC_LC`, `isPSXSPC_LC_uvchr`, and `isPSXSPC_LC_utf8`.
>
> ```
> bool isPSXSPC(char ch)
> ```

isPUNCT

> 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(1) for details. See the top of this section for an explanation of variants `isPUNCT_A`, `isPUNCT_L1`, `isPUNCT_uvchr`, `isPUNCT_utf8`, `isPUNCT_LC`, `isPUNCT_LC_uvchr`, and `isPUNCT_LC_utf8`.
>
> ```
> bool isPUNCT(char ch)
> ```

isSPACE

> 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 variants `isSPACE_A`, `isSPACE_L1`, `isSPACE_uvchr`, `isSPACE_utf8`, `isSPACE_LC`, `isSPACE_LC_uvchr`, and `isSPACE_LC_utf8`.
>
> ```
> bool isSPACE(char ch)
> ```

isUPPER

> 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 variants `isUPPER_A`, `isUPPER_L1`, `isUPPER_uvchr`, `isUPPER_utf8`, `isUPPER_LC`, `isUPPER_LC_uvchr`, and `isUPPER_LC_utf8`.
>
> ```
> bool isUPPER(char ch)
> ```

isWORDCHAR

> 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 variants `isWORDCHAR_A`, `isWORDCHAR_L1`, `isWORDCHAR_uvchr`, and `isWORDCHAR_utf8`. `isWORDCHAR_LC`, `isWORDCHAR_LC_uvchr`, and `isWORDCHAR_LC_utf8` are also as described there, but additionally include the platform's native underscore.

```
        bool isWORDCHAR(char ch)
```

isXDIGIT

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 variants `isXDIGIT_uvchr`, `isXDIGIT_utf8`, `isXDIGIT_LC`, `isXDIGIT_LC_uvchr`, and `isXDIGIT_LC_utf8`.

```
        bool isXDIGIT(char ch)
```

Cloning an interpreter

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;`, reason to keep it around is if you want to dup some of your own variable who are outside the graph perl scans, an example of this code is in *threads.xs* create.

`CLONEf_CLONE_HOST` - This is a win32 thing, it is ignored on unix, it tells perls 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
        )
```

Compile-time scope hooks

BhkDISABLE

NOTE: this function 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: this function 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: this function 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: this function 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: this function must be explicitly called as Perl_blockhook_register with an aTHX_ parameter.

>    ```
>        void Perl_blockhook_register(pTHX_ BHK *hk)
>    ```

COP Hint Hashes

cophh_2hv

>   NOTE: this function 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: this function 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: this function 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(const COPHH *cophh,
>        const char *key, U32 hash,
>        U32 flags)
>    ```

cophh_delete_pvn

>   NOTE: this function 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: this function is experimental and may change or be removed without notice.

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

>    ```
>        COPHH * cophh_delete_pvs(const COPHH *cophh,
>        const char *key, U32 flags)
>    ```

cophh_delete_sv
          NOTE: this function 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(const COPHH *cophh, SV *key,
U32 hash, U32 flags)
```

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

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

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

cophh_fetch_sv
          NOTE: this function 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: this function 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: this function 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: this function 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(const COPHH *cophh,
    const char *key, U32 hash,
    SV *value, U32 flags)
```

cophh_store_pvn

NOTE: this function 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: this function is experimental and may change or be removed without notice.

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

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

cophh_store_sv

NOTE: this function 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(const COPHH *cophh, SV *key,
    U32 hash, SV *value, U32 flags)
```

COP Hint Reading

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_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 NUL-terminated literal string instead of a string/length pair, and no precomputed hash.

```
SV * cop_hints_fetch_pvs(const COP *cop,
const char *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)
```

Custom Operators

custom_op_register

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

NOTE: this function 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)
```

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.

NOTE: this function must be explicitly called as Perl_custom_op_xop with an aTHX_ parameter.

```
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 retreive 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(1) 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 Manipulation Functions

> This section documents functions to manipulate CVs which are code-values, or 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
>  )
> ```

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   Uses `strlen` to get the length of `name`, then calls `get_cvn_flags`.

> NOTE: the perl_ form of this function is deprecated.
>
> ```
> CV* get_cv(const char* name, I32 flags)
> ```

get_cvn_flags

> Returns 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.
>
> NOTE: the perl_ form of this function is deprecated.
>
> ```
> CV* get_cvn_flags(const char* name, STRLEN len,
> I32 flags)
> ```

xsubpp **variables and internal functions**

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;
```

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.

```
(whatever) 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.

```
(whatever) 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.

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.

Debugging Utilities

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_packsubs
> Dumps the optrees for all visible subroutines in `stash`.

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

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 and Dump functions

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)
```

Embedding Functions

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 this function 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 returns the offset in the current pad, or `NOT_IN_PAD`.
>
> Now it always returns `NOT_IN_PAD`.
>
> NOTE: the perl_ form of this function is deprecated.
>
> ```
> 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". flags can be any of `PERL_LOADMOD_DENY`, `PERL_LOADMOD_NOIMPORT`, or `PERL_LOADMOD_IMPORT_OPS` (or 0 for no flags). ver, if specified and not NULL, provides version semantics similar to `use Foo::Bar VERSION`. The optional trailing SV* arguments can be used to specify arguments to the module's `import()` method, similar to `use Foo::Bar VERSION LIST`. They must be terminated with a final `NULL` pointer. Note that this list can only be omitted when the `PERL_LOADMOD_NOIMPORT` flag has been used. Otherwise at least a single `NULL` pointer to designate the default import list is required.
>
> The reference count for each specified SV* parameter is decremented.
>
> ```
> void load_module(U32 flags, SV* name, SV* ver, ...)
> ```

newPADNAMELIST
> NOTE: this function 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: this function 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: this function 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, 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: this function 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: this function 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: this function 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_setsv
>   Set the value at offset `po` in the current (compiling or executing) pad. Use the macro
>   `PAD_SETSV()` rather than calling this function directly.

```
            void pad_setsv(PADOFFSET po, SV *sv)
```

pad_sv   Get the value at offset `po` in the current (compiling or executing) pad. Use macro `PAD_SV` instead of calling this function directly.

```
            SV * pad_sv(PADOFFSET po)
```

pad_tidy NOTE: this function 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_construct
        Initializes a new Perl interpreter. See perlembed.

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

perl_destruct
        Shuts down a Perl interpreter. See perlembed.

```
            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. See perlembed.

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

perl_run Tells a Perl interpreter to run. See perlembed.

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

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_ form of this function is deprecated.

```
            void require_pv(const char* pv)
```

Exception Handling (simple) Macros

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

```
            dXCPT;
```

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.

Functions in file scope.c

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)
```

Functions in file vutil.c

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)
```

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):

```
    SV* vverify(SV *vs)
```

"Gimme" Values
G_ARRAY
            Used to indicate list context. See `"GIMME_V"`, `"GIMME"` and perlcall.

G_DISCARD
            Indicates that arguments returned from a callback should be discarded. See perlcall.

G_EVAL
            Used to force a Perl `eval` wrapper around a callback. See perlcall.

GIMME   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(1) for a usage example.

```
    U32 GIMME_V
```

G_NOARGS

>Indicates that no arguments are being sent to a callback. See perlcall.

G_SCALAR

>Used to indicate scalar context. See `"GIMME_V"`, `"GIMME"`, and perlcall.

G_VOID

>Used to indicate void context. See `"GIMME_V"` and perlcall.

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: this function 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 %ˆH, 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.

GV Functions

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.

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_fetchmeth
        Like "gv_fetchmeth_pvn", but lacks a flags parameter.

    GV* gv_fetchmeth(HV* stash, const char* name,
    STRLEN len, I32 level)

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` and `SVf_UTF8`.

        `GV_SUPER` indicates that we want to look up the method in the superclasses of the `stash`.

        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)
```

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
SVf_UTF8
GV_NOADD_NOINIT
GV_NOINIT
GV_NOEXPAND
GV_ADDMG
```

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 `NUL`-terminated literal string instead of a string/length pair.

```
HV* gv_stashpvs(const char* 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.

```
SV* GvSV(GV* gv)
```

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)
```

Handy Values

Nullav Null AV pointer.

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

Nullch Null character pointer. (No longer available when `PERL_CORE` is defined.)

Nullcv Null CV pointer.

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

Nullhv Null HV pointer.

(deprecated - use `(HV *)NULL` instead)

Nullsv Null SV pointer. (No longer available when `PERL_CORE` is defined.)

Hash Manipulation Functions

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.

cop_fetch_label

NOTE: this function is experimental and may change or be removed without notice.

Returns the label attached to a cop. The flags pointer may be set to `SVf_UTF8` or 0.

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

cop_store_label

NOTE: this function 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.

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

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_ form of this function 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_assert

Check that a hash is in an internally consistent state.

```
                    void hv_assert(HV *hv)
```

hv_clear   Frees the 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 *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(1) 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 `NUL`-terminated literal string instead of a string/length pair.

```
         SV** hv_fetchs(HV* tb, const char* 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(1) for more information on how to use this function on tied hashes.

```
         HE* hv_fetch_ent(HV *hv, SV *keysv, I32 lval,
         U32 hash)
```

hv_fill   Returns the number of hash buckets that happen to be in use. This function is wrapped by the macro `HvFILL`.

Previously this value was always stored in the HV structure, which created an overhead on every hash (and pretty much every object) for something that was rarely used. Now we calculate it on demand the first time that it is needed, and cache it if that calculation is going to be costly to repeat. The cached value is updated by insertions and deletions, but (currently) discarded if the hash is split.

```
         STRLEN hv_fill(HV *const hv)
```

hv_iterinit

Prepares a starting point to traverse a hash table. Returns the number of keys in the hash (i.e. the same as `HvUSEDKEYS(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: this function 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.
>
> ```
>  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. Handles magic when the hash is tied.
>
> ```
>  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(1) 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 `NUL`-terminated literal string instead of a string/length pair and omits the hash parameter.

```
SV** hv_stores(HV* tb, const char* key,
NULLOK 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. `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(1) 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)
```

newHV   Creates a new HV. The reference count is set to 1.

```
HV* newHV()
```

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_pointer` is written into the "PL_check" array, while the value previously stored there is written to `*old_checker_p`.

The function should be defined like this:

```
static OP *new_checker(pTHX_ OP *op) { ... }
```

It is intended to be called in this manner:

```
new_checker(aTHX_ op)
```

old_checker_p should be defined like this:

```
static Perl_check_t 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.

If you want to influence compilation of calls to a specific subroutine, then use "cv_set_call_checker" 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)
```

Lexer interface

This is the lower layer of the Perl parser, managing characters and tokens.

lex_bufutf8

NOTE: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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: this function is experimental and may change or be removed without notice.

        Like "lex_stuff_pvn", but takes a `NUL`-terminated literal string instead of a string/length pair.

```
                void lex_stuff_pvs(const char *pv, U32 flags)
```

lex_stuff_sv
        NOTE: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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: this function 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_termexpr

NOTE: this function 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: this function 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: this function 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: this function 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: this function 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.

Locale-related functions and macros
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
```

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()
```

STORE_LC_NUMERIC_FORCE_TO_UNDERLYING

This is used by XS code that 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.

```
void STORE_LC_NUMERIC_FORCE_TO_UNDERLYING()
```

STORE_LC_NUMERIC_SET_TO_NEEDED

This is used to help wrap XS or C code that that is LC_NUMERIC locale-aware. This locale category is generally kept set to the C locale by Perl for backwards compatibility, and because most XS code that reads floating point values can cope only with the decimal radix character being a dot.

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 uselocale; 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();
...
}
```

```
void STORE_LC_NUMERIC_SET_TO_NEEDED()
```

sync_locale

Changing the program's locale should be avoided by XS code. Nevertheless, certain non-Perl libraries called from XS, such as `Gtk` do so. When this happens, Perl needs to be told that the locale has changed. Use this function to do so, before returning to Perl.

```
void sync_locale()
```

Magical Functions

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_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 this function 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)
```

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)
```

SvLOCK

Arranges for a mutual exclusion lock to be obtained on sv if a suitable module has been loaded.

```
              void SvLOCK(SV* sv)
```

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)
```

SvSetMagicSV

Like SvSetSV, but does any set magic required afterwards.

```
              void SvSetMagicSV(SV* dsv, SV* ssv)
```

SvSetMagicSV_nosteal

Like SvSetSV_nosteal, but does any set magic required afterwards.

```
              void SvSetMagicSV_nosteal(SV* dsv, SV* ssv)
```

SvSetSV

Calls sv_setsv if dsv is not the same as ssv. May evaluate arguments more than once. Does not handle 'set' magic on the destination SV.

```
              void SvSetSV(SV* dsv, SV* ssv)
```

SvSetSV_nosteal

Calls a non-destructive version of sv_setsv if dsv is not the same as ssv. May evaluate arguments more than once.

```
              void SvSetSV_nosteal(SV* dsv, SV* ssv)
```

SvSHARE

Arranges for sv to be shared between threads if a suitable module has been loaded.

```
              void SvSHARE(SV* sv)
```

SvUNLOCK

Releases a mutual exclusion lock on sv if a suitable module has been loaded.

```
              void SvUNLOCK(SV* sv)
```

Memory Management

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)
```

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)
```

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)
```

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)
```

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)
```

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. 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 `"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, I32 len)
```

savepvs   Like `savepvn`, but takes a NUL-terminated literal string instead of a string/length pair.

```
            char* savepvs(const char* s)
```

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(const char* s)
```

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)
```

StructCopy

This is an architecture-independent macro to copy one structure to another.

```
void StructCopy(type *src, type *dest, type)
```

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)
```

Miscellaneous Functions

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)
```

fbm_compile

Analyses 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)
```

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)
```

form   Takes a sprintf-style format pattern and conventional (non-SV) arguments and returns 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);
```

Uses a single 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).

```
char* form(const char* pat, ...)
```

getcwd_sv
        Fill `sv` with current working directory

```
int getcwd_sv(SV* sv)
```

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)
```

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)
```

is_safe_syscall
        Test that the given `pv` doesn't contain any internal `NUL` characters. If it does, set `errno` to
        `ENOENT`, optionally warn, and return FALSE.

        Return TRUE if the name is safe.

        Used by the `IS_SAFE_SYSCALL()` macro.

```
bool is_safe_syscall(const char *pv, STRLEN len,
const char *what,
const char *op_name)
```

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 zero if equal, or non-zero if non-
        equal.

```
bool memEQ(char* s1, char* s2, STRLEN len)
```

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 zero if non-equal, or non-zero
        if equal.

```
bool memNE(char* s1, char* s2, STRLEN len)
```

mess   Take a sprintf-style format pattern and 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''.

        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 * mess(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)
```

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
The C library sprintf, wrapped if necessary, to ensure that it will return the length of the string
written to the buffer. Only rare pre-ANSI systems need the wrapper function - usually this is a
direct call to sprintf.

```
int my_sprintf(char *buffer, 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.

```
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.

```
Size_t my_strlcpy(char *dst, const char *src,
Size_t size)
```

my_vsnprintf
The C library vsnprintf if available and standards-compliant. However, if 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(char * big, char * bigend, char * little,
char * little_end)
```

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()
```

quadmath_format_needed
        `quadmath_format_needed()` returns true if the `format` string seems to contain at least one non-Q-prefixed `%[efgaEFGA]` format specifier, or returns false otherwise.

The format specifier detection is not complete printf-syntax detection, but it should catch most common cases.

If true is returned, those arguments **should** in theory be processed with `quadmath_snprintf()`, but in case there is more than one such format specifier (see "quadmath_format_single"), and if there is anything else beyond that one (even just a single byte), they **cannot** be processed because `quadmath_snprintf()` is very strict, accepting only one format spec, and nothing else. In this case, the code should probably fail.

```
bool quadmath_format_needed(const char* format)
```

quadmath_format_single
        `quadmath_snprintf()` is very strict about its `format` string and will fail, returning -1, if the format is invalid. It accepts exactly one format spec.

quadmath_format_single() checks that the intended single spec looks sane: begins with %, has only one %, ends with [efgaEFGA], and has Q before it. This is not a full "printf syntax check", just the basics.

Returns the format if it is valid, NULL if not.

quadmath_format_single() can and will actually patch in the missing Q, if necessary. In this case it will return the modified copy of the format, **which the caller will need to free.**

See also "quadmath_format_needed".

```
const char* quadmath_format_single(const char* format)
```

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*)
```

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(char * big, char * bigend,
char * little, char * little_end)
```

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)
```

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)
```

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)
```

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)
```

MRO Functions
            These functions are related to the method resolution order of perl classes

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_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(1) for details.

```
            void mro_register(const struct mro_alg *mro)
```

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;
```

Numeric functions

grok_bin

>converts a string representing a binary number to numeric form.

>On entry `start` and `*len` 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 the first invalid character. Unless `PERL_SCAN_SILENT_ILLDIGIT` is set in `*flags`, encountering an invalid character will also trigger a warning. On return `*len` 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 the value to `*result` (or the value 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 the binary number may use `"_"` characters to separate digits.

```
 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 the first invalid character. Unless `PERL_SCAN_SILENT_ILLDIGIT` is set in `*flags`, encountering an invalid character will also trigger a warning. On return `*len` 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 the value to `*result` (or the value 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 the hex number may use `"_"` characters to separate digits.

```
 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_INFINITE
IS_NUMBER_NAN
IS_NUMBER_INFINITE | 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

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` 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 the first invalid character. Unless `PERL_SCAN_SILENT_ILLDIGIT` is set in `*flags`, encountering an 8 or 9 will also trigger a warning. On return `*len` 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 the value to `*result` (or the value is discarded if `result` is NULL).

If `PERL_SCAN_ALLOW_UNDERSCORES` is set in `*flags` then the octal number may use `"_"` characters to separate digits.

```
UV grok_oct(const char* start, STRLEN* len_p,
I32* flags, NV *result)
```

isinfnan   `Perl_isinfnan()` is 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)
```

Perl_signbit

NOTE: this function 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)
```

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)
```

Obsolete backwards compatibility functions

Some of these are also deprecated. You can exclude these from your compiled Perl by adding this option to Configure: `-Accflags='-DNO_MATHOMS'`

custom_op_desc

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

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)
```

gv_fetchmethod
        See "gv_fetchmethod_autoload".

```
 GV* gv_fetchmethod(HV* stash, const char* name)
```

is_utf8_char
        DEPRECATED! It is planned to remove this function 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".

```
 STRLEN is_utf8_char_buf(const U8 *buf,
 const U8 *buf_end)
```

pack_cat
        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)
```

pad_compname_type
        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.

```
 HV * pad_compname_type(PADOFFSET po)
```

sv_2pvbyte_nolen
        Return a pointer to the byte-encoded representation of the SV. May cause the SV to be downgraded from UTF-8 as a side-effect.

        Usually accessed via the SvPVbyte_nolen macro.

```
 char* sv_2pvbyte_nolen(SV* sv)
```

sv_2pvutf8_nolen
        Return a pointer to the UTF-8-encoded representation of the SV. May cause the SV to be upgraded to UTF-8 as a side-effect.

        Usually accessed via the SvPVutf8_nolen macro.

```
 char* sv_2pvutf8_nolen(SV* sv)
```

sv_2pv_nolen
        Like sv_2pv(), but doesn't return the length too. You should usually use the macro wrapper SvPV_nolen(sv) instead.

```
 char* sv_2pv_nolen(SV* sv)
```

sv_catpvn_mg
        Like sv_catpvn, but also handles 'set' magic.

```
              void sv_catpvn_mg(SV *sv, const char *ptr,
              STRLEN len)
```

sv_catsv_mg
        Like `sv_catsv`, but also handles 'set' magic.

```
              void sv_catsv_mg(SV *dsv, SV *ssv)
```

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_iv   A private implementation of the `SvIVx` macro for compilers which can't cope with complex
        macro expressions. Always use the macro instead.

```
               IV sv_iv(SV* sv)
```

sv_nolocking
        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
        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)
```

sv_nv   A private implementation of the `SvNVx` macro for compilers which can't cope with complex
        macro expressions. Always use the macro instead.

```
               NV sv_nv(SV* sv)
```

sv_pv   Use the `SvPV_nolen` macro instead

```
              char* sv_pv(SV *sv)
```

sv_pvbyte
        Use `SvPVbyte_nolen` instead.

```
              char* sv_pvbyte(SV *sv)
```

sv_pvbyten
        A private implementation of the `SvPVbyte` macro for compilers which can't cope with complex
        macro expressions. Always use the macro instead.

```
              char* sv_pvbyten(SV *sv, STRLEN *lp)
```

sv_pvn   A private implementation of the `SvPV` macro for compilers which can't cope with complex
        macro expressions. Always use the macro instead.

```
              char* sv_pvn(SV *sv, STRLEN *lp)
```

sv_pvutf8
        Use the `SvPVutf8_nolen` macro instead

```
              char* sv_pvutf8(SV *sv)
```

sv_pvutf8n
  A private implementation of the `SvPVutf8` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
char* sv_pvutf8n(SV *sv, STRLEN *lp)
```

sv_taint   Taint an SV. Use `SvTAINTED_on` instead.

```
void sv_taint(SV* sv)
```

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_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_mg
  Like `sv_usepvn`, but also handles 'set' magic.

```
void sv_usepvn_mg(SV *sv, char *ptr, STRLEN len)
```

sv_uv   A private implementation of the `SvUVx` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
UV sv_uv(SV* sv)
```

unpack_str
  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.

```
I32 unpack_str(const char *pat, const char *patend,
const char *s, const char *strbeg,
const char *strend, char **new_s,
I32 ocnt, U32 flags)
```

utf8_to_uvchr
  DEPRECATED! It is planned to remove this function 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_uvuni
  DEPRECATED! It is planned to remove this function from a future release of Perl. Do not use it for new code; remove it from existing code.

Returns the Unicode 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 one reason why this function is deprecated. The other is that only in extremely limited circumstances should the Unicode versus native code point be of any interest to you. See "utf8_to_uvuni_buf" for alternatives.

If `s` points to one of the detected malformations, and UTF8 warnings are enabled, zero is returned and `*retlen` is set (if `retlen` doesn't point to 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_uvuni(const U8 *s, STRLEN *retlen)
```

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 that will be locally assigned to a lexical variable, 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 type, 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 *first)
```

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 type, 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, which must have been allocated using `PerlMemShared_malloc`; the memory will be freed

when the op is destroyed.

```
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 *listval)
```

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)
```

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)
```

Optree Manipulation Functions

alloccopstash

NOTE: this function 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

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`, and an SV argument for it is returned in `*ckobj_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` is a GV supplying 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.

By default, the function is Perl_ck_entersub_args_proto_or_list, and the SV parameter is `cv` itself. This implements standard prototype processing. It can be changed, for a particular subroutine, by "cv_set_call_checker".

```
void cv_get_call_checker(CV *cv,
Perl_call_checker *ckfun_p,
SV **ckobj_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.

```
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, and an SV argument for it is supplied in ckobj. 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 flags.

The current setting for a particular CV can be retrieved by "cv_get_call_checker".

```
void cv_set_call_checker_flags(
CV *cv, Perl_call_checker ckfun, SV *ckobj,
U32 flags
)
```

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)
```

newCONSTSUB

See "newCONSTSUB_flags".

```
CV* newCONSTSUB(HV* stash, const char* name, SV* sv)
```

newCONSTSUB_flags

Creates a constant sub equivalent to Perl subFOO(){123} which is eligible for inlining at compile-time.

Currently, the only useful value for flags is SVf_UTF8.

The newly created subroutine takes ownership of a reference to the passed in SV.

Passing `NULL` for SV creates a constant sub equivalent to `subBAR(){}`, which won't be called if used as a destructor, but will suppress the overhead of a call to `AUTOLOAD`. (This form, however, isn't eligible for inlining at compile time.)

```
CV* newCONSTSUB_flags(HV* stash, const char* name,
STRLEN len, U32 flags, SV* sv)
```

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. 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 type, 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. Only use this when an op is no longer linked to from any optree.

```
void op_free(OP *o)
```

OpHAS_SIBLING
Returns true if `o` has a sibling

```
bool OpHAS_SIBLING(OP *o)
```

OpLASTSIB_set

> Marks `o` as having no further siblings. On `PERL_OP_PARENT` builds, 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: this function 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. This function is only available on perls built with `-DPERL_OP_PARENT`.
>
> ```
> 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: this function 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 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 P
>    splice(P, A, 2, X-Y-Z) | | B-C
>    A-B-C-D A-X-Y-Z-D
>
>    P P
>    splice(P, NULL, 1, X-Y) | | A
>    A-B-C-D X-Y-B-C-D
>
>    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

packlist   The engine implementing `pack()` Perl function.

```
    void packlist(SV *cat, const char *pat,
    const char *patend, SV **beglist,
    SV **endlist)
```

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).

```
    I32 unpackstring(const char *pat,
    const char *patend, const char *s,
    const char *strend, U32 flags)
```

Pad Data Structures

CvPADLIST

NOTE: this function 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) }
```

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 NUL-terminated literal string instead of a string/length pair.
>
> ```
> PADOFFSET pad_add_name_pvs(const char *name, U32 flags,
> HV *typestash, HV *ourstash)
> ```

PadARRAY
> NOTE: this function 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 NUL-terminated literal string instead of a string/length pair.
>
> ```
> PADOFFSET pad_findmy_pvs(const char *name, U32 flags)
> ```

PadlistARRAY
>   NOTE: this function 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: this function 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: this function is experimental and may change or be removed without notice.
>
>   The names associated with pad entries.
>
> ```
>  PADNAMELIST * PadlistNAMES(PADLIST padlist)
> ```

PadlistNAMESARRAY
>   NOTE: this function is experimental and may change or be removed without notice.
>
>   The C array of pad names.
>
> ```
>  PADNAME ** PadlistNAMESARRAY(PADLIST padlist)
> ```

PadlistNAMESMAX
>   NOTE: this function is experimental and may change or be removed without notice.
>
>   The index of the last pad name.
>
> ```
>  SSize_t PadlistNAMESMAX(PADLIST padlist)
> ```

PadlistREFCNT
>   NOTE: this function 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: this function is experimental and may change or be removed without notice.
>
>   The index of the last pad entry.
>
> ```
>  SSize_t PadMAX(PAD pad)
> ```

PadnameLEN
>   NOTE: this function is experimental and may change or be removed without notice.
>
>   The length of the name.
>
> ```
>  STRLEN PadnameLEN(PADNAME pn)
> ```

PadnamelistARRAY
>   NOTE: this function is experimental and may change or be removed without notice.
>
>   The C array of pad names.
>
> ```
>  PADNAME ** PadnamelistARRAY(PADNAMELIST pnl)
> ```

PadnamelistMAX
>   NOTE: this function is experimental and may change or be removed without notice.
>
>   The index of the last pad name.

```
               SSize_t PadnamelistMAX(PADNAMELIST pnl)
```

PadnamelistREFCNT
        NOTE: this function 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: this function 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: this function 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: this function 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: this function 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: this function is experimental and may change or be removed without notice.

        Returns the pad name as a mortal SV.

```
          SV * PadnameSV(PADNAME pn)
```

PadnameUTF8
        NOTE: this function 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: this function 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: this function 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: this function is experimental and may change or be removed without notice.

> Points directly to the body of the "PL_comppad" array. (I.e., this is PAD_ARRAY(PL_comppad).)

Per-Interpreter Variables

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.

> ```
> HV* PL_modglobal
> ```

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_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.

> ```
> 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.

> ```
> 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.

```
peep_t PL_rpeepp
```

PL_sv_no
This is the `false` SV. See `"PL_sv_yes"`. Always refer to this as `&PL_sv_no`.

```
SV PL_sv_no
```

PL_sv_undef
This is the `undef` SV. Always refer to this as `&PL_sv_undef`.

```
SV PL_sv_undef
```

PL_sv_yes
This is the `true` SV. See `"PL_sv_no"`. Always refer to this as `&PL_sv_yes`.

```
SV PL_sv_yes
```

REGEXP Functions

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)
```

Stack Manipulation Macros

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;
```

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)
```

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)
```

ORIGMARK

The original stack mark for the XSUB. See `"dORIGMARK"`.

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(1) for other uses.
>
>```
> PUTBACK;
>```

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;
>```

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)
>```

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_mYES

> Place &PL_sv_yes into the specified position pos on the stack.

```
void XST_mYES(int pos)
```

SV-Body Allocation

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)
```

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)
```

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).

> For efficiency, consider using newSVpvn instead.

```
SV* newSVpv(const char *const s, const STRLEN len)
```

newSVpvf

> Creates a new SV and initializes it with the string formatted like sv_catpvf.

```
SV* newSVpvf(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 s, 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)
```

newSVpvs
Like `newSVpvn`, but takes a NUL-terminated literal string instead of a string/length pair.

```
SV* newSVpvs(const char* s)
```

newSVpvs_flags
Like `newSVpvn_flags`, but takes a NUL-terminated literal string instead of a string/length pair.

```
SV* newSVpvs_flags(const char* s, 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 NUL-terminated literal string instead of a string/length pair
and omits the hash parameter.

```
SV* newSVpvs_share(const char* s)
```

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`.

```
SV* newSVrv(SV *const rv,
const char *const classname)
```

newSVsv
Creates a new SV which is an exact duplicate of the original SV. (Uses `sv_setsv`.)

```
SV* newSVsv(SV *const old)
```

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)
```

sv_2bool
This macro is only used by `sv_true()` or its macro equivalent, and only if the latter's argument is neither SvPOK, SvIOK nor SvNOK. It calls `sv_2bool_flags` with the SV_GMAGIC flag.

```
bool sv_2bool(SV *const sv)
```

sv_2bool_flags
This function is only used by `sv_true()` and friends, and only if the latter's argument is neither SvPOK, SvIOK nor SvNOK. If the flags contain SV_GMAGIC, then it does an `mg_get()` first.

```
bool sv_2bool_flags(SV *sv, I32 flags)
```

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
Return a pointer to the byte-encoded representation of the SV, and set `*lp` to its length. May cause the SV to be downgraded from UTF-8 as a side-effect.

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_2pv_flags

Returns a pointer to the string value of an SV, and sets `*lp` to its length. If flags has the `SV_GMAGIC` bit set, does an `mg_get()` first. Coerces `sv` to a string if necessary. Normally invoked via the `SvPV_flags` macro. `sv_2pv()` and `sv_2pv_nomg` usually end up here too.

```
char* sv_2pv_flags(SV *const sv, STRLEN *const lp,
const I32 flags)
```

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

Concatenates the `NUL`-terminated string 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. Handles 'get' magic, but not 'set' magic. See `"sv_catpv_mg"`.

```
void sv_catpv(SV *const sv, const char* ptr)
```

sv_catpvf

Processes its arguments like `sv_catpvfn`, and appends the formatted output to an SV. As with `sv_catpvfn` called with a non-null C-style variable argument list, argument reordering is not supported. 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. Handles 'get' magic, but not 'set' magic. See `"sv_catpvf_mg"`. 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.

```
void sv_catpvf(SV *const sv, const char *const pat,
...)
```

sv_catpvf_mg

Like `sv_catpvf`, but also handles 'set' magic.

```
void sv_catpvf_mg(SV *const sv,
const char *const pat, ...)
```

sv_catpvn

Concatenates the string onto the end of the string which is in the SV. `len` indicates number of bytes to copy. If the SV has the UTF-8 status set, then the bytes appended should be valid UTF-8. Handles 'get' magic, but not 'set' magic. See `"sv_catpvn_mg"`.

```
                void sv_catpvn(SV *dsv, const char *sstr, STRLEN len)
```

sv_catpvn_flags
        Concatenates the string onto the end of the string which is in the SV. The `len` indicates number of bytes to copy.

        By default, 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. One can force the appended string to be interpreted as UTF-8 by supplying the `SV_CATUTF8` flag, and as bytes by supplying the `SV_CATBYTES` flag; the SV or the string appended will be upgraded to UTF-8 if necessary.

        If `flags` has the `SV_SMAGIC` bit set, will `mg_set` on `dsv` afterwards if appropriate. `sv_catpvn` and `sv_catpvn_nomg` are implemented in terms of this function.

```
    void sv_catpvn_flags(SV *const dstr,
    const char *sstr,
    const STRLEN len,
    const I32 flags)
```

sv_catpvs
        Like `sv_catpvn`, but takes a `NUL`-terminated literal string instead of a string/length pair.

```
    void sv_catpvs(SV* sv, const char* s)
```

sv_catpvs_flags
        Like `sv_catpvn_flags`, but takes a `NUL`-terminated literal string instead of a string/length pair.

```
    void sv_catpvs_flags(SV* sv, const char* s,
    I32 flags)
```

sv_catpvs_mg
        Like `sv_catpvn_mg`, but takes a `NUL`-terminated literal string instead of a string/length pair.

```
    void sv_catpvs_mg(SV* sv, const char* s)
```

sv_catpvs_nomg
        Like `sv_catpvn_nomg`, but takes a `NUL`-terminated literal string instead of a string/length pair.

```
    void sv_catpvs_nomg(SV* sv, const char* s)
```

sv_catpv_flags
        Concatenates the `NUL`-terminated string 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. If `flags` has the `SV_SMAGIC` bit set, will `mg_set` on the modified SV if appropriate.

```
    void sv_catpv_flags(SV *dstr, const char *sstr,
    const I32 flags)
```

sv_catpv_mg
        Like `sv_catpv`, but also handles 'set' magic.

```
    void sv_catpv_mg(SV *const sv, const char *const ptr)
```

sv_catsv Concatenates the string from SV `ssv` onto the end of the string in SV `dsv`. If `ssv` is null, does nothing; otherwise modifies only `dsv`. Handles 'get' magic on both SVs, but no 'set' magic. See `"sv_catsv_mg"` and `"sv_catsv_nomg"`.

```
    void sv_catsv(SV *dstr, SV *sstr)
```

sv_catsv_flags
        Concatenates the string from SV `ssv` onto the end of the string in SV `dsv`. If `ssv` is null, does nothing; otherwise modifies only `dsv`. If `flags` has the `SV_GMAGIC` bit set, will call `mg_get` on both SVs if appropriate. If `flags` has the `SV_SMAGIC` bit set, `mg_set` will be called on the modified SV afterward, if appropriate. `sv_catsv`, `sv_catsv_nomg`, and `sv_catsv_mg` are

implemented in terms of this function.

```
void sv_catsv_flags(SV *const dsv, SV *const ssv,
const I32 flags)
```

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 `'usebytes'` 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 `'usebytes'` 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 `'usebytes'` 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 `'usebytes'` 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_flags
       Implementation of `sv_copypv` and `sv_copypv_nomg`. Calls get magic iff flags has the `SV_GMAGIC` bit set.

```
void sv_copypv_flags(SV *const dsv, SV *const ssv,
const I32 flags)
```

sv_copypv_nomg
       Like `sv_copypv`, but doesn't invoke get magic first.

```
void sv_copypv_nomg(SV *const dsv, SV *const ssv)
```

sv_dec   Auto-decrement of the value in the SV, doing string to numeric conversion if necessary. Handles 'get' magic and operator overloading.

```
void sv_dec(SV *const sv)
```

sv_dec_nomg
       Auto-decrement of the value in the SV, doing string to numeric conversion if necessary. Handles operator overloading. Skips handling 'get' magic.

```
void sv_dec_nomg(SV *const sv)
```

sv_eq   Returns a boolean indicating whether the strings in the two SVs are identical. Is UTF-8 and `'usebytes'` 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 `'usebytes'` 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_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)
```

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: this function 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)
```

sv_grow   Expands the character buffer in the SV. If necessary, uses sv_unref and upgrades the SV to
           SVt_PV. Returns a pointer to the character buffer. Use the SvGROW wrapper instead.

```
            char* sv_grow(SV *const sv, STRLEN newlen)
```

sv_inc   Auto-increment of the value in the SV, doing string to numeric conversion if necessary. Handles
           'get' magic and operator overloading.

```
             void sv_inc(SV *const sv)
```

sv_inc_nomg
           Auto-increment of the value in the SV, doing string to numeric conversion if necessary. Handles
           operator overloading. Skips handling 'get' magic.

```
             void sv_inc_nomg(SV *const sv)
```

sv_insert
           Inserts a string at the specified offset/length within the SV. Similar to the Perl substr()
           function. 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 *const little,
            const STRLEN littlelen,
            const U32 flags)
```

sv_isa   Returns a boolean indicating whether the SV is blessed into the specified class. This does not
           check for subtypes; use sv_derived_from to verify an inheritance relationship.

```
          int sv_isa(SV* sv, const char *const name)
```

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)
```

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)
```

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)
```

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)
```

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_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()
```

sv_newref

Increment an SV's reference count. Use the `SvREFCNT_inc()` wrapper instead.

```
        SV* sv_newref(SV *const 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)
```

sv_pvbyten_force

The backend for the `SvPVbytex_force` macro. Always use the macro instead.

```
        char* sv_pvbyten_force(SV *const sv, STRLEN *const lp)
```

sv_pvn_force

Get a sensible string out of the SV somehow. A private implementation of the `SvPV_force` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
        char* sv_pvn_force(SV* sv, STRLEN* lp)
```

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 I32 flags)
```

sv_pvutf8n_force
          The backend for the `SvPVutf8x_force` macro. Always use the macro instead.

```
              char* sv_pvutf8n_force(SV *const sv, STRLEN *const lp)
```

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)
```

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_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)
```

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.

```
              SV* sv_rvweaken(SV *const sv)
```

sv_setiv Copies an integer into the given SV, upgrading first if necessary. Does not handle 'set' magic. See
          also `"sv_setiv_mg"`.

```
              void sv_setiv(SV *const sv, const IV num)
```

sv_setiv_mg
          Like `sv_setiv`, but also handles 'set' magic.

```
              void sv_setiv_mg(SV *const sv, const IV i)
```

sv_setnv Copies a double into the given SV, upgrading first if necessary. Does not handle 'set' magic. See
          also `"sv_setnv_mg"`.

```
              void sv_setnv(SV *const sv, const NV num)
```

sv_setnv_mg
          Like `sv_setnv`, but also handles 'set' magic.

          `void sv_setnv_mg(SV *const sv, const NV num)`

sv_setpv Copies a string into an SV. The string must be terminated with a `NUL` character. Does not handle
          'set' magic. See `"sv_setpv_mg"`.

          `void sv_setpv(SV *const sv, const char *const ptr)`

sv_setpvf
          Works like `sv_catpvf` but copies the text into the SV instead of appending it. Does not handle
          'set' magic. See `"sv_setpvf_mg"`.

          `void sv_setpvf(SV *const sv, const char *const pat,`
          `...)`

sv_setpvf_mg
          Like `sv_setpvf`, but also handles 'set' magic.

          `void sv_setpvf_mg(SV *const sv,`
          `const char *const pat, ...)`

sv_setpviv
          Copies an integer into the given SV, also updating its string value. Does not handle 'set' magic.
          See `"sv_setpviv_mg"`.

          `void sv_setpviv(SV *const sv, const IV num)`

sv_setpviv_mg
          Like `sv_setpviv`, but also handles 'set' magic.

          `void sv_setpviv_mg(SV *const sv, const IV iv)`

sv_setpvn
          Copies 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. Does not handle 'set' magic. See `"sv_setpvn_mg"`.

          `void sv_setpvn(SV *const sv, const char *const ptr,`
          `const STRLEN len)`

sv_setpvn_mg
          Like `sv_setpvn`, but also handles 'set' magic.

          `void sv_setpvn_mg(SV *const sv,`
          `const char *const ptr,`
          `const STRLEN len)`

sv_setpvs
          Like `sv_setpvn`, but takes a `NUL`-terminated literal string instead of a string/length pair.

          `void sv_setpvs(SV* sv, const char* s)`

sv_setpvs_mg
          Like `sv_setpvn_mg`, but takes a `NUL`-terminated literal string instead of a string/length pair.

          `void sv_setpvs_mg(SV* sv, const char* s)`

sv_setpv_mg
          Like `sv_setpv`, but also handles 'set' magic.

          `void sv_setpv_mg(SV *const sv, const char *const ptr)`

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 `NUL`-terminated literal string instead of a string/length pair.

```
SV * sv_setref_pvs(const char* s)
```

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)
```

sv_setsv   Copies the contents of the source SV `ssv` into the destination SV `dsv`. The source SV may be
            destroyed if it is mortal, so don't use this function if the source SV needs to be reused. Does not
            handle 'set' magic on destination SV. Calls 'get' magic on source SV. Loosely speaking, it
            performs a copy-by-value, obliterating any previous content of the destination.

            You probably want to use one of the assortment of wrappers, such as `SvSetSV`,
            `SvSetSV_nosteal`, `SvSetMagicSV` and `SvSetMagicSV_nosteal`.

            ```
            void sv_setsv(SV *dstr, SV *sstr)
            ```

sv_setsv_flags
            Copies the contents of the source SV `ssv` into the destination SV `dsv`. The source SV may be
            destroyed if it is mortal, so don't use this function if the source SV needs to be reused. Does not
            handle 'set' magic. Loosely speaking, it performs a copy-by-value, obliterating any previous
            content of the destination. If the `flags` parameter has the `SV_GMAGIC` bit set, will `mg_get` on
            `ssv` if appropriate, else not. If the `flags` parameter has the `SV_NOSTEAL` bit set then the
            buffers of temps will not be stolen. `sv_setsv` and `sv_setsv_nomg` are implemented in terms
            of this function.

            You probably want to use one of the assortment of wrappers, such as `SvSetSV`,
            `SvSetSV_nosteal`, `SvSetMagicSV` and `SvSetMagicSV_nosteal`.

            This is the primary function for copying scalars, and most other copy-ish functions and macros
            use this underneath.

            ```
            void sv_setsv_flags(SV *dstr, SV *sstr,
            const I32 flags)
            ```

sv_setsv_mg
            Like `sv_setsv`, but also handles 'set' magic.

            ```
            void sv_setsv_mg(SV *const dstr, SV *const sstr)
            ```

sv_setuv   Copies an unsigned integer into the given SV, upgrading first if necessary. Does not handle 'set'
            magic. See also `"sv_setuv_mg"`.

            ```
            void sv_setuv(SV *const sv, const UV num)
            ```

sv_setuv_mg
            Like `sv_setuv`, but also handles 'set' magic.

            ```
            void sv_setuv_mg(SV *const sv, const UV u)
            ```

sv_tainted
            Test an SV for taintedness. Use `SvTAINTED` instead.

            ```
            bool sv_tainted(SV *const sv)
            ```

sv_true   Returns true if the SV has a true value by Perl's rules. Use the `SvTRUE` macro instead, which
            may call `sv_true()` or may instead use an in-line version.

            ```
            I32 sv_true(SV *const 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_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)
```

sv_untaint

        Untaint an SV. Use `SvTAINTED_off` instead.

```
void sv_untaint(SV *const sv)
```

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_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_utf8_decode

        NOTE: this function is experimental and may change or be removed without notice.

        If the PV of the SV is an octet sequence in 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

        NOTE: this function is experimental and may change or be removed without notice.

        Attempts 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, either returns false or, if `fail_ok` is not true, croaks.

        This is not a general purpose Unicode to byte encoding interface: use the `Encode` extension for that.

```
                bool sv_utf8_downgrade(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
        Converts the PV of an SV to its UTF-8-encoded form. Forces the SV to string form if it is not already. Will `mg_get` on `sv` if appropriate. Always sets the `SvUTF8` flag to avoid future validity checks even if the whole string is the same in UTF-8 as not. Returns the number of bytes in the converted string

        This is not a general purpose byte encoding to Unicode interface: use the Encode extension for that.

```
                STRLEN sv_utf8_upgrade(SV *sv)
```

sv_utf8_upgrade_flags
        Converts the PV of an SV to its UTF-8-encoded form. Forces the SV to string form if it is not already. Always sets the SvUTF8 flag to avoid future validity checks even if all the bytes are invariant in UTF-8. If `flags` has `SV_GMAGIC` bit set, will `mg_get` on `sv` if appropriate, else not.

        If `flags` has `SV_FORCE_UTF8_UPGRADE` set, this function assumes that the PV will expand when converted to UTF-8, and skips the extra work of checking for that. Typically this flag is used by a routine that has already parsed the string and found such characters, and passes this information on so that the work doesn't have to be repeated.

        Returns the number of bytes in the converted string.

        This is not a general purpose byte encoding to Unicode interface: use the Encode extension for that.

```
                STRLEN sv_utf8_upgrade_flags(SV *const sv,
                const I32 flags)
```

sv_utf8_upgrade_flags_grow
        Like `sv_utf8_upgrade_flags`, but has an additional parameter `extra`, which is the number of unused bytes the string of `sv` is guaranteed to have free after it upon return. This allows the caller to reserve extra space that it intends to fill, to avoid extra grows.

        `sv_utf8_upgrade`, `sv_utf8_upgrade_nomg`, and `sv_utf8_upgrade_flags` are implemented in terms of this function.

        Returns the number of bytes in the converted string (not including the spares).

```
                STRLEN sv_utf8_upgrade_flags_grow(SV *const sv,
                const I32 flags,
                STRLEN extra)
```

sv_utf8_upgrade_nomg
        Like `sv_utf8_upgrade`, but doesn't do magic on `sv`.

```
                STRLEN sv_utf8_upgrade_nomg(SV *sv)
```

sv_vcatpvf
        Processes its arguments like `sv_catpvfn` called with a non-null C-style variable argument list, and appends the formatted to an SV. Does not handle 'set' magic. See `"sv_vcatpvf_mg"`.

        Usually used via its frontend `sv_catpvf`.

```
                void sv_vcatpvf(SV *const sv, const char *const pat,
                va_list *const args)
```

sv_vcatpvfn

```
                void sv_vcatpvfn(SV *const sv, const char *const pat,
                const STRLEN patlen,
                va_list *const args,
                SV **const svargs, const I32 svmax,
                bool *const maybe_tainted)
```

sv_vcatpvfn_flags

Processes its arguments like `vsprintf` and appends the formatted output to an SV. Uses 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, indicates via `maybe_tainted` if results are untrustworthy (often due to the use of locales).

If called as `sv_vcatpvfn` or flags has the `SV_GMAGIC` bit set, calls get magic.

Usually used via one of its frontends `sv_vcatpvf` and `sv_vcatpvf_mg`.

```
                void sv_vcatpvfn_flags(SV *const sv,
                const char *const pat,
                const STRLEN patlen,
                va_list *const args,
                SV **const svargs,
                const I32 svmax,
                bool *const maybe_tainted,
                const U32 flags)
```

sv_vcatpvf_mg

Like `sv_vcatpvf`, but also handles 'set' magic.

Usually used via its frontend `sv_catpvf_mg`.

```
                void sv_vcatpvf_mg(SV *const sv,
                const char *const pat,
                va_list *const args)
```

sv_vsetpvf

Works like `sv_vcatpvf` but copies the text into the SV instead of appending it. Does not handle 'set' magic. See `"sv_vsetpvf_mg"`.

Usually used via its frontend `sv_setpvf`.

```
                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 I32 svmax,
                bool *const maybe_tainted)
```

sv_vsetpvf_mg
          Like `sv_vsetpvf`, but also handles 'set' magic.

          Usually used via its frontend `sv_setpvf_mg`.

```
 void sv_vsetpvf_mg(SV *const sv,
 const char *const pat,
 va_list *const args)
```

SV Flags

SVt_INVLIST
          Type flag for scalars. See "svtype".

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 Manipulation Functions

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)
```

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_ form of this function is deprecated.

```
     SV* get_sv(const char *name, I32 flags)
```

newRV_inc
          Creates an RV wrapper for an SV. The reference count for the original SV is incremented.

```
     SV* newRV_inc(SV* sv)
```

newSVpadname
          NOTE: this function is experimental and may change or be removed without notice.

          Creates a new SV containing the pad name.

```
     SV* newSVpadname(PADNAME *pn)
```

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(NULLOK const char* s, STRLEN len,
     U32 utf8)
```

sv_catpvn_nomg
          Like `sv_catpvn` but doesn't process magic.

```
     void sv_catpvn_nomg(SV* sv, const char* ptr,
     STRLEN len)
```

sv_catpv_nomg
          Like `sv_catpv` but doesn't process magic.

```
     void sv_catpv_nomg(SV* sv, const char* ptr)
```

sv_catsv_nomg
          Like `sv_catsv` but doesn't process magic.

```
     void sv_catsv_nomg(SV* dsv, SV* ssv)
```

SvCUR   Returns the length of the string which is in the SV. See `"SvLEN"`.

```
     STRLEN SvCUR(SV* sv)
```

SvCUR_set
          Set the current length of the string which is in the SV. See `"SvCUR"` and `SvIV_set>`.

```
     void SvCUR_set(SV* sv, STRLEN len)
```

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.

```
          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)
```

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)
```

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.

```
                        char * SvGROW(SV* sv, STRLEN len)
```

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
          be flagged as either SvUOK or SVIOK.

```
          bool SvIOK_UV(SV* sv)
```

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)
```

SvIV   Coerces the given SV to an integer and returns it. See "SvIVx" for a version which guarantees
          to evaluate sv only once.

```
          IV SvIV(SV* sv)
```

SvIV_nomg
          Like SvIV but doesn't process magic.

```
          IV SvIV_nomg(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)
```

SvIVx   Coerces the given SV to an integer and returns it. Guarantees to evaluate `sv` only once. Only use
         this if `sv` is an expression with side effects, otherwise use the more efficient `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)
```

SvLEN_set
         Set the size of the string buffer for the SV. See `"SvLEN"`.

```
         void SvLEN_set(SV* sv, STRLEN len)
```

SvMAGIC_set
         Set the value of the MAGIC pointer in `sv` to val. See `"SvIV_set"`.

```
         void SvMAGIC_set(SV* sv, MAGIC* val)
```

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)
```

SvNV   Coerce the given SV to a double and return it. See `"SvNVx"` for a version which guarantees to
        evaluate `sv` only once.

            NV SvNV(SV* sv)

SvNV_nomg
        Like `SvNV` but doesn't process magic.

            NV SvNV_nomg(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)

SvNVx   Coerces the given SV to a double and returns it. Guarantees to evaluate `sv` only once. Only use
        this if `sv` is an expression with side effects, otherwise use the more efficient `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_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(NN 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)

SvPV   Returns a pointer to the string in the SV, or a stringified form of the SV if the SV does not contain
>   a string. The SV may cache the stringified version becoming `SvPOK`. Handles 'get' magic. The
>   `len` variable will be set to the length of the string (this is a macro, so don't use `&len`). See also
>   `"SvPVx"` for a version which guarantees to evaluate `sv` only once.
>
>   Note that there is no guarantee that the return value of `SvPV()` is equal to `SvPVX(sv)`, or that
>   `SvPVX(sv)` contains valid data, or that successive calls to `SvPV(sv)` 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"`.
>
>   char* SvPV(SV* sv, STRLEN len)

SvPVbyte
>   Like `SvPV`, but converts `sv` to byte representation first if necessary.
>
>   char* SvPVbyte(SV* sv, STRLEN len)

SvPVbyte_force
>   Like `SvPV_force`, but converts `sv` to byte representation first if necessary.
>
>   char* SvPVbyte_force(SV* sv, STRLEN len)

SvPVbyte_nolen
>   Like `SvPV_nolen`, but converts `sv` to byte representation first if necessary.
>
>   char* SvPVbyte_nolen(SV* sv)

SvPVbytex
>   Like `SvPV`, but converts `sv` to byte representation first if necessary. Guarantees to evaluate `sv`
>   only once; use the more efficient `SvPVbyte` otherwise.
>
>   char* SvPVbytex(SV* sv, STRLEN len)

SvPVbytex_force
>   Like `SvPV_force`, but converts `sv` to byte representation first if necessary. Guarantees to
>   evaluate `sv` only once; use the more efficient `SvPVbyte_force` otherwise.
>
>   char* SvPVbytex_force(SV* sv, STRLEN len)

SvPV_force
>   Like `SvPV` but will force the SV into containing a string (`SvPOK`), and only a string
>   (`SvPOK_only`), by hook or by crook. You need force if you are going to update the `SvPVX`
>   directly. Processes get magic.
>
>   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)"`.
>
>   char* SvPV_force(SV* sv, STRLEN len)

SvPV_force_nomg
>   Like `SvPV_force`, but doesn't process get magic.
>
>   char* SvPV_force_nomg(SV* sv, STRLEN len)

SvPV_nolen
>   Like `SvPV` but doesn't set a length variable.
>
>   char* SvPV_nolen(SV* sv)

SvPV_nomg
>   Like `SvPV` but doesn't process magic.
>
>   char* SvPV_nomg(SV* sv, STRLEN len)

SvPV_nomg_nolen
>   Like `SvPV_nolen` but doesn't process magic.
>
>   char* SvPV_nomg_nolen(SV* sv)

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)

SvPVutf8x
>   Like `SvPV`, but converts `sv` to UTF-8 first if necessary. Guarantees to evaluate `sv` only once; use
>   the more efficient `SvPVutf8` otherwise.
>
>   char* SvPVutf8x(SV* sv, STRLEN len)

SvPVutf8x_force
>   Like `SvPV_force`, but converts `sv` to UTF-8 first if necessary. Guarantees to evaluate `sv` only
>   once; use the more efficient `SvPVutf8_force` otherwise.
>
>   char* SvPVutf8x_force(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)

SvPVX   Returns 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 this macro unless the SV's type >= `SVt_PV`.
>
>   This is also used to store the name of an autoloaded subroutine in an XS AUTOLOAD routine. See
>   "Autoloading with XSUBs" in perlguts.
>
>   char* SvPVX(SV* sv)

SvPVx   A version of `SvPV` which guarantees to evaluate `sv` only once. Only use this if `sv` is an
>   expression with side effects, otherwise use the more efficient `SvPV`.

```
char* SvPVx(SV* sv, STRLEN len)
```

SvREFCNT
        Returns the value of the object's reference count.

```
U32 SvREFCNT(SV* sv)
```

SvREFCNT_dec
        Decrements the reference count of the given SV. sv may be NULL.

```
void SvREFCNT_dec(SV* sv)
```

SvREFCNT_dec_NN
        Same as SvREFCNT_dec, but 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.

```
void SvREFCNT_dec_NN(SV* sv)
```

SvREFCNT_inc
        Increments the reference count of the given SV, returning the SV.

        All of the following SvREFCNT_inc* macros are optimized versions of SvREFCNT_inc, and
        can be replaced with SvREFCNT_inc.

```
SV* SvREFCNT_inc(SV* sv)
```

SvREFCNT_inc_NN
        Same as SvREFCNT_inc, but 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.

```
SV* SvREFCNT_inc_NN(SV* sv)
```

SvREFCNT_inc_simple
        Same as SvREFCNT_inc, but can only be used with expressions without side effects. Since we
        don't have to store a temporary value, it's faster.

```
SV* SvREFCNT_inc_simple(SV* sv)
```

SvREFCNT_inc_simple_NN
        Same as SvREFCNT_inc_simple, but 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.

```
SV* SvREFCNT_inc_simple_NN(SV* sv)
```

SvREFCNT_inc_simple_void
        Same as SvREFCNT_inc_simple, but can only be used if you don't need the return value.
        The macro doesn't need to return a meaningful value.

```
void SvREFCNT_inc_simple_void(SV* sv)
```

SvREFCNT_inc_simple_void_NN
        Same as SvREFCNT_inc, but can only be used if you 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.

```
void SvREFCNT_inc_simple_void_NN(SV* sv)
```

SvREFCNT_inc_void
        Same as SvREFCNT_inc, but can only be used if you don't need the return value. The macro
        doesn't need to return a meaningful value.

```
void SvREFCNT_inc_void(SV* sv)
```

SvREFCNT_inc_void_NN
        Same as SvREFCNT_inc, but can only be used if you 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.

```
                  void SvREFCNT_inc_void_NN(SV* sv)
```

sv_report_used
        Dump the contents of all SVs not yet freed (debugging aid).

```
        void sv_report_used()
```

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_setsv_nomg
        Like sv_setsv but doesn't process magic.

```
        void sv_setsv_nomg(SV* dsv, SV* ssv)
```

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
> Returns a boolean indicating whether Perl would evaluate the SV as true or false. See `"SvOK"` for a defined/undefined test. Handles 'get' magic unless the scalar is already `SvPOK`, `SvIOK` or `SvNOK` (the public, not the private flags).
>
>   bool SvTRUE(SV* sv)

SvTRUE_nomg
> Returns a boolean indicating whether Perl would evaluate the SV as true or false. See `"SvOK"` for a defined/undefined test. Does not handle 'get' magic.
>
>   bool SvTRUE_nomg(SV* sv)

SvTYPE
> Returns the type of the SV. See `"svtype"`.
>
>   svtype SvTYPE(SV* sv)

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 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)

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_upgrade_nomg
> Like `sv_utf8_upgrade`, but doesn't do magic on `sv`.
>
>   STRLEN sv_utf8_upgrade_nomg(NN SV *sv)

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   Coerces the given SV to an unsigned integer and returns it. See `"SvUVx"` for a version which guarantees to evaluate `sv` only once.

>   UV SvUV(SV* sv)

SvUV_nomg
> Like `SvUV` but doesn't process magic.
>
>   UV SvUV_nomg(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)
> ```

SvUVx Coerces the given SV to an unsigned integer and returns it. Guarantees to evaluate `sv` only once.
       Only use this if `sv` is an expression with side effects, otherwise use the more efficient `SvUV`.

> ```
> UV SvUVx(SV* sv)
> ```

SvVOK Returns a boolean indicating whether the SV contains a v-string.

> ```
> bool SvVOK(SV* sv)
> ```

Unicode Support

"Unicode Support" in perlguts(1) has an introduction to this API.

See also "Character classification", and "Character case changing". Various functions outside this section
also work specially with Unicode. Search for the string "utf8" in this document.

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: this function is experimental and may change or be removed without notice.
>
> Converts a string `s` of length `len` from UTF-8 into native byte encoding. Unlike "utf8_to_bytes"
> but like "bytes_to_utf8", returns a pointer to the newly-created string, and updates `len` to
> contain the new length. Returns the original string if no conversion occurs, `len` is unchanged. Do
> nothing if `is_utf8` points to 0. Sets `is_utf8` to 0 if `s` is converted or consisted entirely of
> characters that are invariant in UTF-8 (i.e., US-ASCII on non-EBCDIC machines).
>
> ```
> U8* bytes_from_utf8(const U8 *s, STRLEN *len,
> bool *is_utf8)
> ```

bytes_to_utf8

> NOTE: this function is experimental and may change or be removed without notice.
>
> Converts a string `s` of length `len` bytes from the native encoding into UTF-8. Returns a pointer
> to the newly-created string, and sets `len` to reflect the new length in bytes.
>
> 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 *len)
> ```

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 <http://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_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_invariant_string` is
preferred.

```
    bool is_ascii_string(const U8 *s, STRLEN len)
```

is_invariant_string

        Returns true iff 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). That is, 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 and `\c?` (which isn't in the ASCII range on EBCDIC).

        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_loclen"(), and "is_utf8_string_loc"().

```
    bool is_invariant_string(const U8 *s, STRLEN len)
```

is_utf8_string

        Returns true if the first `len` bytes of string `s` form a valid UTF-8 string, 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'.

See also "is_invariant_string"(), "is_utf8_string_loclen"(), and "is_utf8_string_loc"().

```
bool is_utf8_string(const U8 *s, STRLEN len)
```

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.

See also "is_utf8_string_loclen"() and "is_utf8_string"().

```
bool is_utf8_string_loc(const U8 *s, 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, and the number of UTF-8 encoded characters in the el.

See also "is_utf8_string_loc"() and "is_utf8_string"().

```
bool is_utf8_string_loclen(const U8 *s, STRLEN len,
const U8 **ep, STRLEN *el)
```

isUTF8_CHAR

Returns the number of bytes beginning at s which form a legal UTF-8 (or UTF-EBCDIC) encoded character, looking no further than e-s bytes into s. Returns 0 if the sequence starting at s through e-1 is not well-formed UTF-8.

Note that an INVARIANT character (i.e. ASCII on non-EBCDIC machines) is a valid UTF-8 character.

```
STRLEN isUTF8_CHAR(const U8 *s, const U8 *e)
```

pv_uni_display

Build to the scalar dsv a displayable version of the 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.

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)
```

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)
```

to_utf8_case

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

Instead use the appropriate one of "toUPPER_utf8", "toTITLE_utf8", "toLOWER_utf8", or "toFOLD_utf8".

p contains the pointer to the UTF-8 string encoding the character that is being converted. This routine assumes that the character at p is well-formed.

ustrp is a pointer to the character buffer to put the conversion result to. lenp is a pointer to the length of the result.

swashp is a pointer to the swash to use.

Both the special and normal mappings are stored in *lib/unicore/To/Foo.pl*, and loaded by SWASHNEW, using *lib/utf8_heavy.pl*. special (usually, but not always, a multicharacter mapping), is tried first.

special is a string, normally NULL or "". NULL means to not use any special mappings; "" means to use the special mappings. Values other than these two are treated as the name of the hash containing the special mappings, like "utf8::ToSpecLower".

normal is a string like "ToLower" which means the swash %utf8::ToLower.

Code points above the platform's IV_MAX will raise a deprecation warning, unless those are turned off.

```
     UV to_utf8_case(const U8 *p, U8* ustrp,
     STRLEN *lenp, SV **swashp,
     const char *normal,
     const char *special)
```

to_utf8_fold

Instead use "toFOLD_utf8".

```
     UV to_utf8_fold(const U8 *p, U8* ustrp,
     STRLEN *lenp)
```

to_utf8_lower
        Instead use "toLOWER_utf8".

```
UV to_utf8_lower(const U8 *p, U8* ustrp,
STRLEN *lenp)
```

to_utf8_title
        Instead use "toTITLE_utf8".

```
UV to_utf8_title(const U8 *p, U8* ustrp,
STRLEN *lenp)
```

to_utf8_upper
        Instead use "toUPPER_utf8".

```
UV to_utf8_upper(const U8 *p, U8* ustrp,
STRLEN *lenp)
```

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, when a malformation is found, zero is 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.

        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. For allowed 0 length
        strings, this function returns 0; for allowed overlong sequences, the computed code point is
        returned; for all other allowed malformations, the Unicode REPLACEMENT CHARACTER is
        returned, as these have no determinable reasonable value.

        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. 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.

        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. 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.

        The flags UTF8_WARN_ILLEGAL_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.)

It is now deprecated to have very high code points (above IV_MAX on the platforms) and this function will raise a deprecation warning for these (unless such warnings are turned off). This value, is typically 0x7FFF_FFFF (2**31 -1) in a 32-bit word.

Code points above 0x7FFF_FFFF (2**31 - 1) were never specified in any standard, so using them is more problematic than other above-Unicode code points. Perl invented an extension to UTF-8 to represent the ones above 2**36-1, so it is likely that non-Perl languages will not be able to read files that contain these that written by the perl interpreter; 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. These are the UTF8_WARN_ABOVE_31_BIT and UTF8_DISALLOW_ABOVE_31_BIT flags. These are entirely independent from the deprecation warning for code points above IV_MAX. On 32-bit machines, it will eventually be forbidden to have any code point that needs more than 31 bits to represent. When that happens, effectively the UTF8_DISALLOW_ABOVE_31_BIT flag will always be set on 32-bit machines. (Of course UTF8_DISALLOW_SUPER will treat all above-Unicode code points, including these, as malformations; and UTF8_WARN_SUPER warns on these.)

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. The flags UTF8_WARN_ABOVE_31_BIT and UTF8_DISALLOW_ABOVE_31_BIT have the same function as on ASCII platforms, warning and disallowing 2**31 and higher.

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, U32 flags)
```

utf8n_to_uvuni
  Instead use "utf8_to_uvchr_buf", or rarely, "utf8n_to_uvchr".

This function was useful for code that wanted to handle both EBCDIC and ASCII platforms with Unicode properties, but starting in Perl v5.20, the distinctions between the platforms have mostly been made invisible to most code, so this function is quite unlikely to be what you want. If you do need this precise functionality, use instead NATIVE_TO_UNI(utf8_to_uvchr_buf(...)) or NATIVE_TO_UNI(utf8n_to_uvchr(...)).

```
UV utf8n_to_uvuni(const U8 *s, STRLEN curlen,
STRLEN *retlen, U32 flags)
```

UTF8SKIP
  returns the number of bytes in the UTF-8 encoded character whose first (perhaps only) byte is pointed to by s.

```
STRLEN UTF8SKIP(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_length

        Return the length of the UTF-8 char encoded string s in characters. Stops at e (inclusive). If e < s or if the scan would end up past e, croaks.

```
  STRLEN utf8_length(const U8* s, const U8 *e)
```

utf8_to_bytes

        NOTE: this function is experimental and may change or be removed without notice.

        Converts a string s of length len from UTF-8 into native byte encoding. Unlike "bytes_to_utf8", this over-writes the original string, and updates len to contain the new length. Returns zero on failure, setting len to -1.

        If you need a copy of the string, see "bytes_from_utf8".

```
  U8* utf8_to_bytes(U8 *s, STRLEN *len)
```

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.

        Code points above the platform's IV_MAX will raise a deprecation warning, unless those are turned off.

```
  UV utf8_to_uvchr_buf(const U8 *s, const U8 *send,
  STRLEN *retlen)
```

utf8_to_uvuni_buf

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

        Only in very rare circumstances should code need to be dealing in Unicode (as opposed to native) code points. In those few cases, use NATIVE_TO_UNI(utf8_to_uvchr_buf(...)) instead.

        Returns the Unicode (not-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.

Code points above the platform's `IV_MAX` will raise a deprecation warning, unless those are turned off.

```
UV utf8_to_uvuni_buf(const U8 *s, const U8 *send,
STRLEN *retlen)
```

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 UV as input, but very high code points (above `IV_MAX` on the platform) will raise a deprecation warning. This 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 UV as input, but very high code points (above `IV_MAX` for the platform) will raise a deprecation warning. This 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.

Code points above 0x7FFF_FFFF (2**31 - 1) were never specified in any standard, so using them is more problematic than other above-Unicode code points. Perl invented an extension to UTF-8 to represent the ones above 2**36-1, so it is likely that non-Perl languages will not be able to read files that contain these that written by the perl interpreter; 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. These are the `UNICODE_WARN_ABOVE_31_BIT` and `UNICODE_DISALLOW_ABOVE_31_BIT` flags. These are entirely independent from the deprecation warning for code points above `IV_MAX`. On 32-bit machines, it will eventually be forbidden to have any code point that needs more than 31 bits to represent. When that happens, effectively the `UNICODE_DISALLOW_ABOVE_31_BIT` flag will always be set on 32-bit machines. (Of course `UNICODE_DISALLOW_SUPER` will treat all above-Unicode code points, including these, as malformations; and `UNICODE_WARN_SUPER` warns on these.)

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. The flags `UNICODE_WARN_ABOVE_31_BIT` and `UNICODE_DISALLOW_ABOVE_31_BIT` have the same function as on ASCII platforms, warning and disallowing 2**31 and higher.

```
U8* uvchr_to_utf8_flags(U8 *d, UV uv, UV flags)
```

uvoffuni_to_utf8_flags

THIS FUNCTION SHOULD BE USED IN ONLY VERY SPECIALIZED CIRCUMSTANCES. Instead, **Almost all code should use "uvchr_to_utf8" or "uvchr_to_utf8_flags"**.

This function is like them, but the input is a strict Unicode (as opposed to native) code point. Only in very rare circumstances should code not be using the native code point.

For details, see the description for "uvchr_to_utf8_flags".

```
U8* uvoffuni_to_utf8_flags(U8 *d, UV uv, UV flags)
```

uvuni_to_utf8_flags

Instead you almost certainly want to use "uvchr_to_utf8" or "uvchr_to_utf8_flags".

This function is a deprecated synonym for "uvoffuni_to_utf8_flags", which itself, while not deprecated, should be used only in isolated circumstances. These functions were useful for code that wanted to handle both EBCDIC and ASCII platforms with Unicode properties, but starting in Perl v5.20, the distinctions between the platforms have mostly been made invisible to most code, so this function is quite unlikely to be what you want.

```
U8* uvuni_to_utf8_flags(U8 *d, UV uv, UV flags)
```

Variables created by `xsubpp` and `xsubpp` internal functions

newXSproto

Used by `xsubpp` to hook up XSUBs as Perl subs. Adds Perl prototypes to the subs.

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;
```

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;
```

Warning and Dieing

croak   This is an XS interface to Perl's `die` function.

Take a sprintf-style format pattern and 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 croak(const char *pat, ...)
```

croak_no_modify
Exactly equivalent to `Perl_croak(aTHX_ "%s", PL_no_modify)`, but generates terser object code than using `Perl_croak`. 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.

```
OP * die(const char *pat, ...)
```

die_sv   Behaves the same as "croak_sv", except for the return type. It should be used only where the `OP *` return type is required. The function never actually returns.

```
OP * die_sv(SV *baseex)
```

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.

> `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 or object will by default be written to standard error, but this is subject to modification by a $SIG{__WARN__} handler.
>
> Unlike with "vcroak", `pat` is not permitted to be null.
>
> ```
> void vwarn(const char *pat, va_list *args)
> ```

warn   This is an XS interface to Perl's `warn` function.

> Take a sprintf-style format pattern and 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 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.
>
> ```
> void warn(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)
> ```

Undocumented functions

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.

GetVars
Gv_AMupdate
PerlIO_clearerr

PerlIO_close
PerlIO_context_layers
PerlIO_eof
PerlIO_error
PerlIO_fileno
PerlIO_fill
PerlIO_flush
PerlIO_get_base
PerlIO_get_bufsiz
PerlIO_get_cnt
PerlIO_get_ptr
PerlIO_read
PerlIO_seek
PerlIO_set_cnt
PerlIO_set_ptrcnt
PerlIO_setlinebuf
PerlIO_stderr
PerlIO_stdin
PerlIO_stdout
PerlIO_tell
PerlIO_unread
PerlIO_write
amagic_call
amagic_deref_call
any_dup
atfork_lock
atfork_unlock
av_arylen_p
av_iter_p
block_gimme
call_atexit
call_list
calloc
cast_i32
cast_iv
cast_ulong
cast_uv
ck_warner
ck_warner_d
ckwarn
ckwarn_d
clear_defarray
clone_params_del
clone_params_new
croak_memory_wrap
croak_nocontext
csighandler
cx_dump
cx_dup
cxinc
deb
deb_nocontext

debop
debprofdump
debstack
debstackptrs
delimcpy
despatch_signals
die_nocontext
dirp_dup
do_aspawn
do_binmode
do_close
do_gv_dump
do_gvgv_dump
do_hv_dump
do_join
do_magic_dump
do_op_dump
do_open
do_open9
do_openn
do_pmop_dump
do_spawn
do_spawn_nowait
do_sprintf
do_sv_dump
doing_taint
doref
dounwind
dowantarray
dump_eval
dump_form
dump_indent
dump_mstats
dump_sub
dump_vindent
filter_add
filter_del
filter_read
foldEQ_latin1
form_nocontext
fp_dup
fprintf_nocontext
free_global_struct
free_tmps
get_context
get_mstats
get_op_descs
get_op_names
get_ppaddr
get_vtbl
gp_dup
gp_free

gp_ref
gv_AVadd
gv_HVadd
gv_IOadd
gv_SVadd
gv_add_by_type
gv_autoload4
gv_autoload_pv
gv_autoload_pvn
gv_autoload_sv
gv_check
gv_dump
gv_efullname
gv_efullname3
gv_efullname4
gv_fetchfile
gv_fetchfile_flags
gv_fetchpv
gv_fetchpvn_flags
gv_fetchsv
gv_fullname
gv_fullname3
gv_fullname4
gv_handler
gv_name_set
he_dup
hek_dup
hv_common
hv_common_key_len
hv_delayfree_ent
hv_eiter_p
hv_eiter_set
hv_free_ent
hv_ksplit
hv_name_set
hv_placeholders_get
hv_placeholders_set
hv_rand_set
hv_riter_p
hv_riter_set
ibcmp_utf8
init_global_struct
init_stacks
init_tm
instr
is_lvalue_sub
leave_scope
load_module_nocontext
magic_dump
malloc
markstack_grow
mess_nocontext

mfree
mg_dup
mg_size
mini_mktime
moreswitches
mro_get_from_name
mro_get_private_data
mro_set_mro
mro_set_private_data
my_atof
my_atof2
my_bcopy
my_bzero
my_chsize
my_cxt_index
my_cxt_init
my_dirfd
my_exit
my_failure_exit
my_fflush_all
my_fork
my_lstat
my_memcmp
my_memset
my_pclose
my_popen
my_popen_list
my_setenv
my_socketpair
my_stat
my_strftime
newANONATTRSUB
newANONHASH
newANONLIST
newANONSUB
newATTRSUB
newAVREF
newCVREF
newFORM
newGVREF
newGVgen
newGVgen_flags
newHVREF
newHVhv
newIO
newMYSUB
newPROG
newRV
newSUB
newSVREF
newSVpvf_nocontext
new_stackinfo

op_refcnt_lock
op_refcnt_unlock
parser_dup
perl_alloc_using
perl_clone_using
pmop_dump
pop_scope
pregcomp
pregexec
pregfree
pregfree2
printf_nocontext
ptr_table_fetch
ptr_table_free
ptr_table_new
ptr_table_split
ptr_table_store
push_scope
re_compile
re_dup_guts
re_intuit_start
re_intuit_string
realloc
reentrant_free
reentrant_init
reentrant_retry
reentrant_size
ref
reg_named_buff_all
reg_named_buff_exists
reg_named_buff_fetch
reg_named_buff_firstkey
reg_named_buff_nextkey
reg_named_buff_scalar
regdump
regdupe_internal
regexec_flags
regfree_internal
reginitcolors
regnext
repeatcpy
rsignal
rsignal_state
runops_debug
runops_standard
rvpv_dup
safesyscalloc
safesysfree
safesysmalloc
safesysrealloc
save_I16
save_I32

save_I8
save_adelete
save_aelem
save_aelem_flags
save_alloc
save_aptr
save_ary
save_bool
save_clearsv
save_delete
save_destructor
save_destructor_x
save_freeop
save_freepv
save_freesv
save_generic_pvref
save_generic_svref
save_hash
save_hdelete
save_helem
save_helem_flags
save_hints
save_hptr
save_int
save_item
save_iv
save_list
save_long
save_mortalizesv
save_nogv
save_op
save_padsv_and_mortalize
save_pptr
save_pushi32ptr
save_pushptr
save_pushptrptr
save_re_context
save_scalar
save_set_svflags
save_shared_pvref
save_sptr
save_svref
save_vptr
savestack_grow
savestack_grow_cnt
scan_num
scan_vstring
seed
set_context
set_numeric_local
set_numeric_radix
set_numeric_standard

```
share_hek
si_dup
ss_dup
stack_grow
start_subparse
str_to_version
sv_2iv
sv_2pv
sv_2uv
sv_catpvf_mg_nocontext
sv_catpvf_nocontext
sv_dup
sv_dup_inc
sv_peek
sv_pvn_nomg
sv_setpvf_mg_nocontext
sv_setpvf_nocontext
sys_init
sys_init3
sys_intern_clear
sys_intern_dup
sys_intern_init
sys_term
taint_env
taint_proper
unlnk
unsharepvn
utf16_to_utf8
utf16_to_utf8_reversed
uvuni_to_utf8
vdeb
vform
vload_module
vnewSVpvf
vwarner
warn_nocontext
warner
warner_nocontext
whichsig
whichsig_pv
whichsig_pvn
whichsig_sv
```

AUTHORS

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

perlguts(1), perlxs(1), perlxstut(1), perlintern(1)