

## 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. The interfaces of any functions that are not listed here are subject to change without notice. For this reason, blindly using functions listed in `proto.h` is to be avoided when writing extensions.

Note that all Perl API global variables must be referenced with the `PL_` prefix. 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.

Note that Perl can be compiled and run under EBCDIC (See *perlebcdic*) or ASCII. 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 more bytes than in UTF-8.

Also, on some EBCDIC machines, functions that are documented as operating on US-ASCII (or Basic Latin in Unicode terminology) may in fact operate on all 256 characters in the EBCDIC range, not just the subset corresponding to US-ASCII.

The listing below is alphabetical, case insensitive.

## "Gimme" Values

### 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.

U32 GIMME\_V

### G\_ARRAY

Used to indicate list context. See `GIMME_V`, `GIMME` and *percall*.

### G\_DISCARD

Indicates that arguments returned from a callback should be discarded. See *percall*.

### G\_EVAL

Used to force a Perl `eval` wrapper around a callback. See *percall*.

### G\_NOARGS

Indicates that no arguments are being sent to a callback. See *percall*.

### G\_SCALAR

Used to indicate scalar context. See `GIMME_V`, `GIMME`, and *percall*.

`G_VOID`

Used to indicate void context. See `GIMME_V` and *percall*.

## Array Manipulation Functions

`AvFILL`

Same as `av_len()`. Deprecated, use `av_len()` instead.

```
int AvFILL(AV* av)
```

`av_clear`

Clears an array, making it empty. Does not free the memory used by the array itself.

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

`av_create_and_push`

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

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

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

`av_create_and_unshift_one`

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

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

```
SV** av_create_and_unshift_one(AV **const avp, SV *const val)
```

`av_delete`

Deletes the element indexed by `key` from the array. Returns the deleted element. If `flags` equals `G_DISCARD`, the element is freed and null is returned.

```
SV* av_delete(AV *av, I32 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 `&PL_sv_undef`.

```
bool av_exists(AV *av, I32 key)
```

`av_extend`

Pre-extend an array. The `key` is the index to which the array should be extended.

```
void av_extend(AV *av, I32 key)
```

`av_fetch`

Returns the SV at the specified index in the array. The `key` is the index. If `lval` is set then the fetch will be part of a store. Check that the return value is non-null before dereferencing it to a `SV*`.

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

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

**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 an array will be `fill + 1` after `av_fill()` returns. If the array was previously shorter then the additional elements appended are set to `PL_sv_undef`. 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, I32 fill)
```

**av\_len**

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

```
I32 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.

```
AV* av_make(I32 size, SV **strp)
```

**av\_pop**

Pops an SV off the end of the array. Returns `&PL_sv_undef` if the array is empty.

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

**av\_push**

Pushes an SV onto the end of the array. The array will grow automatically to accommodate the addition. Like `av_store`, this takes ownership of one reference count.

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

**av\_shift**

Shifts an SV off the beginning of the array. Returns `&PL_sv_undef` if the array is empty.

```
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 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`.

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

```
SV** av_store(AV *av, I32 key, SV *val)
```

**av\_undef**

Undefines the array. Frees the memory used by the array itself.

```
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.

```
void av_unshift(AV *av, I32 num)
```

### get\_av

Returns the AV of the specified Perl array. `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.

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

### newAV

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

```
AV* newAV()
```

### sortsv

Sort an array. Here is an example:

```
sortsv(AvARRAY(av), av_len(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 Perl sub. See *percall*.

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 *percall*.

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 *percall*.

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 whose name is in the SV. See *percall*.

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 *percall*.

```
ENTER;
```

**eval\_pv**

Tells Perl to `eval` the given string 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.

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 *percall*.

```
FREETMPS;
```

**LEAVE**

Closing bracket on a callback. See `ENTER` and *percall*.

```
LEAVE;
```

**SAVETMPS**

Opening bracket for temporaries on a callback. See `FREETMPS` and *percall*.

```
SAVETMPS;
```

**Character classes****isALNUM**

Returns a boolean indicating whether the C `char` is a US-ASCII (Basic Latin) alphanumeric character (including underscore) or digit.

```
bool isALNUM(char ch)
```

**isALPHA**

Returns a boolean indicating whether the C `char` is a US-ASCII (Basic Latin) alphabetic character.

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

**isDIGIT**

Returns a boolean indicating whether the C `char` is a US-ASCII (Basic Latin) digit.

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

**isLOWER**

Returns a boolean indicating whether the C `char` is a US-ASCII (Basic Latin) lowercase character.

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

**isSPACE**

Returns a boolean indicating whether the C `char` is a US-ASCII (Basic Latin) whitespace.

```
bool isSPACE(char ch)
```

**isUPPER**

Returns a boolean indicating whether the C `char` is a US-ASCII (Basic Latin) uppercase character.

```
bool isUPPER(char ch)
```

**toLOWER**

Converts the specified character to lowercase. Characters outside the US-ASCII (Basic Latin) range are viewed as not having any case.

```
char toLOWER(char ch)
```

**toUPPER**

Converts the specified character to uppercase. Characters outside the US-ASCII (Basic Latin) range are viewed as not having any case.

```
char toUPPER(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, 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)
```

## CV Manipulation Functions

### CvSTASH

Returns the stash of the CV.

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

### 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)
```

## Embedding Functions

### 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)
```

### 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, 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.

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

### nothreadhook

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

```
int nothreadhook()
```

### perl\_alloc

Allocates a new Perl interpreter. See *perlembed*.

```
PerlInterpreter* perl_alloc()
```

### perl\_construct

Initializes a new Perl interpreter. See *perlembded*.

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

#### perl\_destruct

Shuts down a Perl interpreter. See *perlembded*.

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

#### perl\_free

Releases a Perl interpreter. See *perlembded*.

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

#### perl\_parse

Tells a Perl interpreter to parse a Perl script. See *perlembded*.

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

#### perl\_run

Tells a Perl interpreter to run. See *perlembded*.

```
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)
```

## Functions in file `dump.c`

#### 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.

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 Unicode, if



PERL\_PV\_ESCAPE\_UNI\_DETECT is set then the input string is scanned using `is_utf8_string()` to determine if it is Unicode.

If PERL\_PV\_ESCAPE\_ALL is set then all input chars will be output using `\x01F1` style escapes, otherwise only chars above 255 will be escaped using this style, other non printable chars will use octal or common escaped patterns like `\n`. 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 string is utf8 and the chars value is `>255` 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 3 or more digit 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)
```

## Functions in file mathoms.c

### gv\_fetchmethod

See `gv_fetchmethod_autoload`.

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

### 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)
```

### 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)
```

## Functions in file `perl.h`

### 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()
```

## Functions in file `pp_ctl.c`

### 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. (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)
```

## Functions in file `pp_pack.c`

### 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 `unpack()` Perl function. `unpackstring` puts the extracted list items on the stack and returns the number of elements. Issue `PUTBACK` before and `SPAGAIN` after the call to this function.

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

Functions in file `pp_sys.c`

## 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)
```

## Global Variables

## PL\_keyword\_plugin

Function pointer, pointing at a function used to handle extended keywords. The function should be declared as

```
int keyword_plugin_function(pTHX_
char *keyword_ptr, STRLEN keyword_len,
OP **op_ptr)
```

The function is called from the tokeniser, whenever a possible keyword is seen. `keyword_ptr` points at the word in the parser's input buffer, and `keyword_len` gives its length; it is not null-terminated. The function is expected to examine the word, and possibly other state such as `%^H`, to decide whether it wants to handle it as an extended keyword. If it does not, the function should return `KEYWORD_PLUGIN_DECLINE`, and the normal parser process will continue.

If the function wants to handle the keyword, it first must parse anything following the keyword that is part of the syntax introduced by the keyword. See *Lexer interface* for details.

When a keyword is being handled, the plugin function must build a tree of `OP` structures, representing the code that was parsed. The root of the tree must be stored in `*op_ptr`. The function then returns a constant indicating the syntactic role of the construct that it has parsed: `KEYWORD_PLUGIN_STMT` if it is a complete statement, or `KEYWORD_PLUGIN_EXPR` if it is an expression. Note that a statement construct cannot be used inside an expression (except via `do BLOCK` and similar), and an expression is not a complete statement (it requires at least a terminating semicolon).

When a keyword is handled, the plugin function may also have (compile-time) side effects. It may modify `%^H`, define functions, and so on. Typically, if side effects are the main purpose of a handler, it does not wish to generate any ops to be included in the normal compilation. In this case it is still required to supply an op tree, but it suffices to generate a single null op.

That's how the `*PL_keyword_plugin` function needs to behave overall. Conventionally, however, one does not completely replace the existing handler function. Instead, take a copy of `PL_keyword_plugin` before assigning your own function pointer to it. Your handler function should look for keywords that it is interested in and handle those. Where it is not interested, it should call the saved plugin function, passing on the arguments it received. Thus `PL_keyword_plugin` actually points at a chain of handler functions, all of which have an opportunity to

handle keywords, and only the last function in the chain (built into the Perl core) will normally return `KEYWORD_PLUGIN_DECLINE`.

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

## GV Functions

### GvSV

Return the SV from the GV.

```
SV* GvSV(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)
```

### gv\_fetchmeth

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.

This function grants "SUPER" token as a postfix of the stash name. 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(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 via a side effect to do this.

These functions have the same side-effects and as `gv_fetchmeth` with `level==0`. `name` should be writable if contains `':'` or `' '`. The warning against passing the GV returned by `gv_fetchmeth` to `call_sv` apply equally to these functions.

```
GV* gv_fetchmethod_autoload(HV* stash, const char* name, I32 autoload)
```

### gv\_fetchmeth\_autoload

Same as `gv_fetchmeth()`, but looks for autoloaded subroutines too. Returns a glob for the subroutine.

For an autoloader subroutine without a GV, will create a GV even if `level < 0`. For an autoloader subroutine without a stub, `GvCV()` of the result may be zero.

```
HV* gv_fetchmeth_autoload(HV* stash, const char* name, STRLEN
len, I32 level)
```

#### 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.

```
HV* gv_stashpvn(const char* name, U32 namelen, I32 flags)
```

#### gv\_stashpvs

Like `gv_stashpvn`, but takes a literal string instead of a string/length pair.

```
HV* gv_stashpvs(const char* name, I32 create)
```

#### gv\_stashsv

Returns a pointer to the stash for a specified package. See `gv_stashpvn`.

```
HV* gv_stashsv(SV* sv, I32 flags)
```

## 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

#### 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`.

```
char* HeUTF8(HE* he)
```

#### HeVAL

Returns the value slot (type `SV*`) stored in the hash entry.

```
SV* HeVAL(HE* he)
```

#### HvNAME

Returns the package name of a stash, or `NULL` if `stash` isn't a stash. See `SvSTASH`, `CvSTASH`.

```
char* HvNAME(HV* stash)
```

#### hv\_assert

Check that a hash is in an internally consistent state.

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

#### hv\_clear

Clears a hash, making it empty.

```
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\_delete

Deletes a key/value pair in the hash. The value `SV` is removed from the hash and returned to the caller. The `klen` is the length of the key. The `flags` value will normally be zero; if set to `G_DISCARD` then `NULL` will be returned.

```
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 and returned to the caller. The `flags` value will normally be zero; if set to `G_DISCARD` then `NULL` will be returned. `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)
```

#### hv\_exists

Returns a boolean indicating whether the specified hash key exists. The `klen` is the length of the key.

```
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 `klen` is the length of the key. If `lval` is set then the fetch will be part of a store. Check that the return value is non-null before dereferencing it to an SV\*.

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

```
SV** hv_fetch(HV *hv, const char *key, I32 klen, I32 lval)
```

#### hv\_fetchs

Like `hv_fetch`, but takes a literal string instead of a string/length pair.

```
SV** hv_fetchs(HV* tb, 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 `tb` 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 perlguides* 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\_iterinit

Prepares a starting point to traverse a hash table. Returns the number of keys in the hash (i.e. the same as `HvKEYS(tb)`). 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(tb)`.

```
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_interval` in one operation.

```
SV* hv_iternextsv(HV *hv, char **key, I32 *retlen)
```

### hv\_iternext\_flags

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 `&Perl_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.

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

```
HE* hv_iternext_flags(HV *hv, I32 flags)
```

### hv\_interval

Returns the value from the current position of the hash iterator. See `hv_iterkey`.

```
SV* hv_interval(HV *hv, HE *entry)
```

### hv\_magic

Adds magic to a hash. See `sv_magic`.

```
void hv_magic(HV *hv, GV *gv, int how)
```

### 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 `klen` is the length of the key. 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 `SV`s then store them in a hash, `hv_store` will own the only reference to the new `SV`, and your code doesn't need to do anything further to tidy up. `hv_store` is not implemented as a call to `hv_store_ent`, and does not create a temporary `SV` for the key, so if your key data is not already in `SV` form then use `hv_store` in preference to `hv_store_ent`.

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

```
SV** hv_store(HV *hv, const char *key, I32 klen, SV *val, U32 hash)
```

#### hv\_stores

Like `hv_store`, but takes a literal string instead of a string/length pair and omits the hash parameter.

```
SV** hv_stores(HV* tb, 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 `SV`s 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* 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.

```
void hv_undef(HV *hv)
```

#### newHV

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

```
HV* newHV()
```

## Lexer interface

### lex\_bufutf8

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.

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

```
bool lex_bufutf8()
```

### lex\_discard\_to

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 necessary 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.

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

```
void lex_discard_to(char *ptr)
```

### lex\_grow\_linestr

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.

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

```
char * lex_grow_linestr(STRLEN len)
```

### lex\_next\_chunk

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* includes `LEX_KEEP_PREVIOUS`, 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.

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

```
bool lex_next_chunk(U32 flags)
```

#### lex\_peek\_unichar

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* includes `LEX_KEEP_PREVIOUS` 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.

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

```
I32 lex_peek_unichar(U32 flags)
```

#### lex\_read\_space

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* includes `LEX_KEEP_PREVIOUS` then the current chunk will not be discarded.

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

```
void lex_read_space(U32 flags)
```

#### lex\_read\_to

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*.

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

```
void lex_read_to(char *ptr)
```

#### lex\_read\_unichar

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* includes `LEX_KEEP_PREVIOUS` 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.

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

```
I32 lex_read_unichar(U32 flags)
```

### lex\_stuff\_pvn

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 interpreted is available as a Perl scalar, the *lex\_stuff\_sv* function is more convenient.

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

```
void lex_stuff_pvn(char *pv, STRLEN len, U32 flags)
```

### lex\_stuff\_sv

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 interpreted is not already a Perl scalar, the *lex\_stuff\_pvn* function avoids the need to construct a scalar.

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

```
void lex_stuff_sv(SV *sv, U32 flags)
```

### lex\_unstuff

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*.

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

```
void lex_unstuff(char *ptr)
```

### 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

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.

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

#### PL\_parser->bufptr

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`.

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

#### PL\_parser->linestart

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.

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

#### PL\_parser->linestr

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.

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

## 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\_free**

Free any magic storage used by the SV. See `sv_magic`.

```
int mg_free(SV* sv)
```

**mg\_get**

Do magic after a value is retrieved from the SV. See `sv_magic`.

```
int mg_get(SV* sv)
```

**mg\_length**

Report on the SV's length. See `sv_magic`.

```
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. 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 macro evaluates its argument more than once.

```
void SvSETMAGIC(SV* sv)
```

**SvSetMagicSV**

Like `SvSetSV`, but does any set magic required afterwards.

```
void SvSetMagicSV(SV* dsb, 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.

```
void SvSetSV(SV* dsb, 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.

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 *perlhack*). 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`.

```
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`.

```
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.

```
void Renew(void* ptr, int nitems, type)
```

**Renewc**

The XSUB-writer's interface to the C `realloc` function, with cast.

```
void Renewc(void* ptr, int nitems, type, cast)
```

**Safefree**

The XSUB-writer's interface to the C `free` function.

```
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()`. The memory allocated for the new string can be freed with the `Safefree()` function.

```
char* savepv(const char* pv)
```

**savepv**

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.

```
char* savepv(const char* pv, I32 len)
```

**savepvs**

Like `savepv`, but takes a 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)
```

**savesharedpv**

A version of `savepv()` which allocates the duplicate string in memory which is shared between threads. (With the specific difference that a NULL pointer is not acceptable)

```
char* savesharedpv(const char *const pv, const STRLEN len)
```

**savesvpv**

A version of `savepv()/savepv()` which gets the string to duplicate from the passed in SV using `SvPV()`

```
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

**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 *str* and *strend*. 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)
```

#### form

Takes a printf-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 the *sv* with current working directory

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

#### 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\_vsnprintf

The C library `vsnprintf` if available and standards-compliant. However, if the `vsnprintf` is not available, will unfortunately use the unsafe `vsprintf` which can overrun the buffer (there is an overrun check, but that may be too late). Consider using `sv_vcatpvf` instead, or getting `vsnprintf`.

```
int my_vsnprintf(char *buffer, const Size_t len, const char
*format, va_list ap)
```

#### 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

```
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)
```

#### strEQ

Test two strings to see if they are equal. Returns true or false.

```
bool strEQ(char* s1, char* s2)
```

#### strGE

Test two 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 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 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 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 strings to see if they are different. Returns true or false.

```
bool strNE(char* s1, char* s2)
```

**strnEQ**

Test two 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 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)
```

**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.

```
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.

```
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

```
SV* vstringify(SV *vs)
```

### vverify

Validates that the SV contains a valid version object.

```
bool vverify(SV *vobj);
```

Note that it only confirms the bare minimum structure (so as not to get confused by derived classes which may contain additional hash entries):

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

## MRO Functions

### mro\_get\_linear\_isa

Returns either `mro_get_linear_isa_c3` or `mro_get_linear_isa_dfs` for the given stash, dependant upon which MRO is in effect for that 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)
```

## Multicall Functions

### dMULTICALL

Declare local variables for a multicall. See *"Lightweight Callbacks" in perlcall*.

```
dMULTICALL;
```



**MULTICALL**

Make a lightweight callback. See *"Lightweight Callbacks" in percall*.

```
MULTICALL;
```

**POP\_MULTICALL**

Closing bracket for a lightweight callback. See *"Lightweight Callbacks" in percall*.

```
POP_MULTICALL;
```

**PUSH\_MULTICALL**

Opening bracket for a lightweight callback. See *"Lightweight Callbacks" in percall*.

```
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  $\leq$  `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* 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  $\leq$  `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\_number

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 UV, it is returned in the `*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.

```
int grok_number(const char *pv, STRLEN len, UV *valuep)
```

## 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 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  $\leq$  `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)
```

## Perl\_signbit

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. As a first pass, this gets everything right except `-0.0`. Alas, catching `-0.0` is the main use for this function, so this is not too helpful yet. Still, at least we have the scaffolding in place to support other systems, should that prove useful.

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()`.

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

```
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)
```

## Optree Manipulation Functions

`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)
```

`newCONSTSUB`

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

Passing `NULL` for `SV` creates a constant sub equivalent to `sub BAR () {}`, 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(HV* stash, const char* name, 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.

## Pad Data Structures

`pad_findmy`

Given a lexical name, try to find its offset, first in the current pad, or failing that, in the pads of any lexically enclosing subs (including the complications introduced by `eval`). If the name is found in an outer pad, then a fake entry is added to the current pad. Returns the offset in the current pad, or `NOT_IN_PAD` on failure.

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

```
PADOFFSET pad_findmy(const char* name, STRLEN len, U32 flags)
```

`pad_sv`

Get the value at offset `po` in the current pad. Use macro `PAD_SV` instead of calling this function directly.

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

## 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\_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) &&
    (tmpsv = (SV*)SvRV(sv)) &&
    SvTYPE(tmpsv) == SVt_PVMG &&
```

```

        (tmpmg = mg_find(tmpsv, PERL_MAGIC_qr))
    {
        return (REGEXP *)tmpmg->mg_obj;
    }

```

NULL will be returned if a REGEXP\* is not found.

```
REGEXP * SvRX(SV *sv)
```

#### SvRXOK

Returns a boolean indicating whether the SV contains qr magic (PERL\_MAGIC\_qr). If you want to do something with the REGEXP\* later use SvRX instead and check for NULL.

```
bool SvRXOK(SV* sv)
```

## Simple Exception Handling Macros

#### dXCPT

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

```
dXCPT;
```

#### XCPT\_CATCH

Introduces a catch block. See *"Exception Handling" in perlguits*.

#### XCPT\_RETHROW

Rethrows a previously caught exception. See *"Exception Handling" in perlguits*.

```
XCPT_RETHROW;
```

#### XCPT\_TRY\_END

Ends a try block. See *"Exception Handling" in perlguits*.

#### XCPT\_TRY\_START

Starts a try block. See *"Exception Handling" in perlguits*.

## 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, int 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
```

**POPi**

Pops a long off the stack.

```
long POPl
```

**POPn**

Pops a double off the stack.

```
NV POPn
```

**POPp**

Pops a string off the stack. Deprecated. New code should use POPpx.

```
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.

```
char* POPpx
```

**POPs**

Pops an SV off the stack.

```
SV* POPs
```

**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 *percall*.

```
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 *percall* 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 *percall*.



```
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 Flags

### 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.

### SVt\_IV

Integer type flag for scalars. See `svtype`.

### SVt\_NV

Double type flag for scalars. See `svtype`.

### SVt\_PV

Pointer type flag for scalars. See `svtype`.

### SVt\_PVAV

Type flag for arrays. See `svtype`.

### SVt\_PVCV

Type flag for code refs. See `svtype`.

### SVt\_PVHV

Type flag for hashes. See `svtype`.

### SVt\_PVMG

Type flag for blessed scalars. See `svtype`.

## SV Manipulation Functions

### 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 %s::%s(%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)
```

### newSVpvn\_utf8

Creates a new SV and copies a string 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)
```

### 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)
```

### SvEND

Returns a pointer to the last character in the string which is in the SV. See SvCUR. Access the character as \*(SvEND(sv)).

```
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.

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

### SvIOK

Returns a U32 value indicating whether the SV contains an integer.

```
U32 SvIOK(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\_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 and SV that it is an unsigned integer and disables all other OK bits.

```
void SvIOK_only_UV(SV* sv)
```

#### SvIOK\_UV

Returns a boolean indicating whether the SV contains an unsigned integer.

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

#### SvIsCOW

Returns a boolean 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)

```
bool 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)
```

#### 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)
```

### 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)
```

### 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 actual length of the string which is in the SV. See `SvIV_set`.

```
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)
```

### 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)
```

### SvNIOK\_off

Unsets the NV/IV status of an SV.

```
void SvNIOK_off(SV* sv)
```

### SvNOK

Returns a U32 value indicating whether the SV contains a double.

```
U32 SvNOK(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)
```

### 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)
```

### 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)
```

### 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)
```

### SvNV\_set

Set the value of the NV pointer in `sv` to `val`. See `SvIV_set`.

```
void SvNV_set(SV* sv, NV val)
```

### 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)
```

### SvPOKp

Returns a U32 value indicating whether the SV contains a character string. Checks the **private** setting. Use SvPOK instead.

```
U32 SvPOKp(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)
```

### 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. See also SvPVx for a version which guarantees to evaluate sv only once.

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



**SvPVbyte**

Like `SvPV`, but converts `sv` to byte representation first if necessary.

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

**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)
```

**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)
```

**SvPVutf8**

Like `SvPV`, but converts `sv` to utf8 first if necessary.

```
char* SvPVutf8(SV* sv, STRLEN len)
```

**SvPVutf8x**

Like `SvPV`, but converts `sv` to utf8 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 utf8 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 utf8 first if necessary.

```
char* SvPVutf8_force(SV* sv, STRLEN len)
```

**SvPVutf8\_nolen**

Like `SvPV_nolen`, but converts `sv` to utf8 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.

```
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 `SvPVX`.

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

#### SvPV\_force

Like `SvPV` but will force the SV into containing just a string (`SvPOK_only`). You want force if you are going to update the `SvPVX` directly.

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

#### SvPV\_force\_nomg

Like `SvPV` but will force the SV into containing just a string (`SvPOK_only`). You want force if you are going to update the `SvPVX` directly. Doesn't process magic.

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

#### SvPV\_nolen

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 form becoming `SvPOK`. Handles 'get' magic.

```
char* SvPV_nolen(SV* sv)
```

#### SvPV\_nomg

Like `SvPV` but doesn't process magic.

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

#### SvPV\_set

Set the value of the PV pointer in `sv` to `val`. See `SvIV_set`.

```
void SvPV_set(SV* sv, char* val)
```

#### SvREFCNT

Returns the value of the object's reference count.

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

#### SvREFCNT\_dec

Decrements the reference count of the given SV.

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

#### SvREFCNT\_inc

Increments the reference count of the given 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)
```

**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)
```

## 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.

```
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. Does not handle 'get' magic.

```
bool SvTRUE(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 unsigned integer.

```
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 whether the SV contains UTF-8 encoded data. Call this after `SvPV()` in case any call to string overloading updates the internal flag.

```
U32 SvUTF8(SV* sv)
```

**SvUTF8\_off**

Unsets the UTF-8 status of an SV.

```
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)
```

**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 `sv` only once. Only use this if `sv` is an expression with side effects, otherwise use the more efficient `SvUV`.

```
UV SvUVx(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)
```

## SvVOK

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

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

## sv\_catpvn\_nomg

Like `sv_catpvn` but doesn't process magic.

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

## sv\_catsv\_nomg

Like `sv_catsv` but doesn't process magic.

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

## sv\_derived\_from

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.

```
bool sv_derived_from(SV* sv, const char *const name)
```

## sv\_does

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, const char *const name)
```

## sv\_report\_used

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

```
void sv_report_used()
```

## sv\_setsv\_nomg

Like `sv_setsv` but doesn't process magic.

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

## sv\_utf8\_upgrade\_nomg

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

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

## 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.

```
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 sv)
```

### 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 *perlhack*). 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 the 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 into it. The reference count for the SV is set to 1. If `len` is zero, Perl will compute the length using `strlen()`. 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 `sprintf`.

```
SV* newSVpvf(const char *const pat, ...)
```

### newSVpvn

Creates a new SV and copies a string 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.

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

### newSVpvn\_flags

Creates a new SV and copies a string 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 `sv2mortal()` 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 `READONLY` and `FAKE`. If the `hash` parameter is non-zero, that value is used; otherwise the hash is computed. The string's hash can be 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 literal string instead of a string/length pair.

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

### newSVpvs\_flags

Like `newSVpvn_flags`, but takes a literal string instead of a string/length pair.

```
SV* newSVpvs_flags(const char* s, U32 flags)
```

### newSVpvs\_share

Like `newSVpvn_share`, but takes a 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 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.

```
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)
```

### 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)
```



**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)
```

**sv\_2bool**

This function is only called on magical items, and is only used by `sv_true()` or its macro equivalent.

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

**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.

```
IO* sv_2io(SV *const sv)
```

**sv\_2iv\_flags**

Return the integer value of an SV, doing any necessary string conversion. If flags includes `SV_GMAGIC`, 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**

Return the num value of an SV, doing any necessary string or integer conversion, magic etc. Normally used via the `SvNV(sv)` and `SvNVx(sv)` macros.

```
NV sv_2nv(SV *const sv)
```

**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 *const 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 *const 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` includes `SV_GMAGIC`, 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` includes `SV_GMAGIC`, 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.

```
int 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 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 `sprintf` and appends the formatted output to an SV. 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. The `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. If the SV has the UTF-8 status set, then the bytes appended should be valid UTF-8. If `flags` has `SV_GMAGIC` bit set, will `mg_get` on `dsv` if appropriate, else not. `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 literal string instead of a string/length pair.

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

### 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`. Modifies `dsv` but not `ssv`. Handles 'get' magic, but not 'set' magic. See `sv_catsv_mg`.

```
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`. Modifies `dsv` but not `ssv`. If `flags` has `SV_GMAGIC` bit set, will `mg_get` on the SVs if appropriate, else not. `sv_catsv` and `sv_catsv_nomg` 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)` must be true and the `ptr` must be a pointer to somewhere inside the string buffer. The `ptr` becomes the first character of the adjusted string. Uses the "OOK hack". Beware: after this function returns, `ptr` and `SvPVX_const(sv)` may no longer refer to the same chunk of data.

```
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 sv)
```

#### sv\_cmp

Compares the strings in two SVs. Returns -1, 0, or 1 indicating whether the string in `sv1` is less than, equal to, or greater than the string in `sv2`. Is UTF-8 and 'use bytes' aware, handles get magic, and will coerce its args to strings if necessary. See also `sv_cmp_locale`.

```
I32 sv_cmp(SV *const sv1, SV *const sv2)
```

#### sv\_cmp\_locale

Compares the strings in two SVs in a locale-aware manner. Is UTF-8 and 'use bytes' aware, handles get magic, and will coerce its args to strings if necessary. See also `sv_cmp`.

```
I32 sv_cmp_locale(SV *const sv1, SV *const sv2)
```

#### sv\_collxfrm

Add Collate Transform magic to an SV if it doesn't already have it.

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(SV *const sv, STRLEN *const npx)
```

#### sv\_copypv

Copies a stringified representation of the source SV into the destination SV. Automatically performs any necessary `mg_get` and coercion of numeric values into strings. Guaranteed to preserve UTF8 flag even from overloaded objects. Similar in nature to `sv_2pv[_flags]` but operates directly on an SV instead of just the string. Mostly uses `sv_2pv_flags` to do its work, except when that would lose the UTF-8'ness of the PV.

```
void sv_copypv(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.

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

#### sv\_eq

Returns a boolean indicating whether the strings in the two SVs are identical. Is UTF-8 and 'use bytes' aware, handles get magic, and will coerce its args to strings if necessary.

```
I32 sv_eq(SV* sv1, SV* sv2)
```

#### sv\_force\_normal\_flags

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`; 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 `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 unrefing. `sv_force_normal` calls this function with `flags` set to 0.

```
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, deallocate 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.

```
char* sv_gets(SV *const sv, PerlIO *const fp, I32 append)
```

#### 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.

```
void sv_inc(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 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. 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 `savepv` 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.

```
void sv_pos_b2u(SV *const sv, I32 *const offsetp)
```

#### 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 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 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 `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\_reftype

Returns a string describing what the SV is a reference to.

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

```
void sv_setpvs(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\_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. 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 `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\_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. You generally want to use the `SvUPGRADE` macro wrapper. 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. The `ptr` should point to memory that was allocated by `malloc`. The string length, `len`, must be supplied. By default this function will `realloc` (i.e. 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

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 being off. Scans PV for validity and returns false if the PV is invalid UTF-8.

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

```
bool sv_utf8_decode(SV *const sv)
```

#### sv\_utf8\_downgrade

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 as a general purpose Unicode to byte encoding interface: use the Encode extension for that.

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

```
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 as 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. Returns the number of bytes in the converted string  
`sv_utf8_upgrade` and `sv_utf8_upgrade_nomg` are implemented in terms of this function.

This is not as 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_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 `vsprintf` and appends the formatted output 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`

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). When running with taint checks enabled, indicates via `maybe_tainted` if results are untrustworthy (often due to the use of locales).

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

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

#### `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)
```

## Unicode Support

### bytes\_from\_utf8

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 utf8 (i.e., US-ASCII on non-EBCDIC machines).

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

```
U8* bytes_from_utf8(const U8 *s, STRLEN *len, bool *is_utf8)
```

### bytes\_to\_utf8

Converts a string `s` of length `len` from the native encoding into UTF-8. Returns a pointer to the newly-created string, and sets `len` to reflect the new length.

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()`.

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

```
U8* bytes_to_utf8(const U8 *s, STRLEN *len)
```

### ibcmp\_utf8

Return true if the strings `s1` and `s2` differ case-insensitively, false if not (if they are equal case-insensitively). If `u1` is true, the string `s1` is assumed to be in UTF-8-encoded Unicode. If `u2` is true, the string `s2` is assumed to be in UTF-8-encoded Unicode. If `u1` or `u2` are false, the respective string is assumed to be in native 8-bit encoding.

If the `pe1` and `pe2` are non-NULL, the scanning pointers will be copied in there (they will point at the beginning of the *next* character). If the pointers behind `pe1` or `pe2` are non-NULL, they are the end pointers beyond which scanning will not continue under any circumstances. If the byte lengths `l1` and `l2` are non-zero, `s1+l1` and `s2+l2` will be used as goal end pointers that will also stop the scan, and which qualify towards

defining a successful match: all the scans that define an explicit length must reach their goal pointers for a match to succeed).

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 ibcmp_utf8(const char *s1, char **pe1, UV l1, bool u1,
const char *s2, char **pe2, UV l2, bool u2)
```

### is\_ascii\_string

Returns true if first `len` bytes of the given string are ASCII (i.e. none of them even raise the question of UTF-8-ness).

See also `is_utf8_string()`, `is_utf8_string_loclen()`, and `is_utf8_string_loc()`.

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

### is\_utf8\_char

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.

```
STRLEN is_utf8_char(const U8 *s)
```

### is\_utf8\_string

Returns true if first `len` bytes of the given string form a valid UTF-8 string, false otherwise. Note that 'a valid UTF-8 string' does not mean 'a string that contains code points above 0x7F encoded in UTF-8' because a valid ASCII string is a valid UTF-8 string.

See also `is_ascii_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 **p)
```

### 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 `e1`.

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

```
bool is_utf8_string_loclen(const U8 *s, STRLEN len, const U8
**ep, STRLEN *e1)
```

### pv\_uni\_display

Build to the scalar `dsv` a displayable version of the string `spv`, length `len`, the displayable version being at most `pvlm` 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.

```
char* pv_uni_display(SV *dsv, const U8 *spv, STRLEN len, STRLEN
pvlm, UV flags)
```

#### sv\_cat\_decode

The encoding is assumed to be an `Encode` object, the PV of the ssv is assumed to be octets in that encoding and decoding the input starts from the position which (PV + \*offset) pointed to. The dsv will be concatenated 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 the ssv. The value which the offset points will be modified to the last input position on the 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

The encoding is assumed to be an `Encode` object, on entry the PV of the sv is assumed to be octets in that encoding, and the sv will be converted into Unicode (and UTF-8).

If the sv already is UTF-8 (or if it is not POK), or if the encoding is not a reference, nothing is done to the sv. If the encoding is not an `Encode::XS` `Encode` object, bad things will happen. (See *lib/encoding.pm* and *Encode*).

The PV of the 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 pvlm 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 pvlm, UV flags)
```

#### to\_utf8\_case

The "p" contains the pointer to the UTF-8 string encoding the character that is being converted.

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

The "swashp" is a pointer to the swash to use.

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

The "special" is a string like `"utf8::ToSpecLower"`, which means the hash `%utf8::ToSpecLower`. The access to the hash is through `Perl_to_utf8_case()`.



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

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

#### to\_utf8\_fold

Convert the UTF-8 encoded character at *p* to its foldcase version and store that in UTF-8 in *ustrp* and its length in bytes in *lenp*. Note that the *ustrp* needs to be at least UTF8\_MAXBYTES\_CASE+1 bytes since the foldcase version may be longer than the original character (up to three characters).

The first character of the foldcased version is returned (but note, as explained above, that there may be more.)

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

#### to\_utf8\_lower

Convert the UTF-8 encoded character at *p* to its lowercase version and store that in UTF-8 in *ustrp* and its length in bytes in *lenp*. Note that the *ustrp* needs to be at least UTF8\_MAXBYTES\_CASE+1 bytes since the lowercase version may be longer than the original character.

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

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

#### to\_utf8\_title

Convert the UTF-8 encoded character at *p* to its titlecase version and store that in UTF-8 in *ustrp* and its length in bytes in *lenp*. Note that the *ustrp* needs to be at least UTF8\_MAXBYTES\_CASE+1 bytes since the titlecase version may be longer than the original character.

The first character of the titlecased version is returned (but note, as explained above, that there may be more.)

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

#### to\_utf8\_upper

Convert the UTF-8 encoded character at *p* to its uppercase version and store that in UTF-8 in *ustrp* and its length in bytes in *lenp*. Note that the *ustrp* needs to be at least UTF8\_MAXBYTES\_CASE+1 bytes since the uppercase version may be longer than the original character.

The first character of the uppercased version is returned (but note, as explained above, that there may be more.)

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

#### utf8n\_to\_uvchr

##### flags

Returns the native character value 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.

Allows length and flags to be passed to low level routine.

```
UV utf8n_to_uvchr(const U8 *s, STRLEN curlen, STRLEN *retlen,
U32 flags)
```

### utf8n\_to\_uvuni

Bottom level UTF-8 decode routine. Returns the Unicode code point value of the first character in the string *s* which is assumed to be in UTF-8 encoding and no longer than *curlen*; *retlen* will be set to the length, in bytes, of that character.

If *s* does not point to a well-formed UTF-8 character, the behaviour is dependent on the value of *flags*: if it contains `UTF8_CHECK_ONLY`, it is assumed that the caller will raise a warning, and this function will silently just set *retlen* to `-1` and return zero. If the *flags* does not contain `UTF8_CHECK_ONLY`, warnings about malformations will be given, *retlen* will be set to the expected length of the UTF-8 character in bytes, and zero will be returned.

The *flags* can also contain various flags to allow deviations from the strict UTF-8 encoding (see *utf8.h*).

Most code should use `utf8_to_uvchr()` rather than call this directly.

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

### 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, I32 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

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`.

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

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

### utf8\_to\_uvchr

Returns the native character value 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.

If *s* does not point to a well-formed UTF-8 character, zero is returned and *retlen* is set, if possible, to `-1`.

```
UV utf8_to_uvchr(const U8 *s, STRLEN *retlen)
```

### utf8\_to\_uvuni

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.

This function should only be used when the returned UV is considered an index into the Unicode semantic tables (e.g. swashes).

If *s* does not point to a well-formed UTF-8 character, zero is returned and *retlen* is set, if possible, to -1.

```
UV utf8_to_uvuni(const U8 *s, STRLEN *retlen)
```

### uvchr\_to\_utf8

Adds the UTF-8 representation of the Native codepoint *uv* to the end of the string *d*; *d* should be have at least `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;
```

```
U8* uvchr_to_utf8(U8 *d, UV uv)
```

### uvuni\_to\_utf8\_flags

Adds the UTF-8 representation of the Unicode codepoint *uv* to the end of the string *d*; *d* should be have at least `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 = uvuni_to_utf8_flags(d, uv, flags);
```

or, in most cases,

```
d = uvuni_to_utf8(d, uv);
```

(which is equivalent to)

```
d = uvuni_to_utf8_flags(d, uv, 0);
```

is the recommended Unicode-aware way of saying

```
*(d++) = uv;
```

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

## Variables created by xsubpp and xsubpp internal functions

### ax

Variable which is setup by `xsubpp` to indicate the stack base offset, used by the `ST`, `XSPrePUSH` and `XSPRETURN` 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 the `padoff_du` variable for an XSUB that wishes to use `UNDERBAR`.

```
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
```

#### newXSproto

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

#### 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`.

## 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 the XSUB-writer's interface to Perl's `die` function. Normally call this function the same way you call the C `printf` function. Calling `croak` returns control directly to Perl, sidestepping the normal C order of execution. See `warn`.

If you want to throw an exception object, assign the object to `$@` and then pass `NULL` to `croak()`:

```
errsv = get_sv("@", GV_ADD);  
sv_setsv(errsv, exception_object);  
croak(NULL);
```

```
void croak(const char* pat, ...)
```

### warn

This is the XSUB-writer's interface to Perl's `warn` function. Call this function the same way you call the C `printf` function. See `croak`.

```
void warn(const char* pat, ...)
```

## Undocumented functions

These functions are currently undocumented:

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

Slab\_Alloc

Slab\_Free

amagic\_call

any\_dup

apply\_attrs\_string

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  
croak\_nocontext  
csighandler  
custom\_op\_desc  
custom\_op\_name  
cx\_dump  
cx\_dup  
cxinc  
deb  
deb\_nocontext  
debop  
debprofdump  
debstack  
debstackptrs  
delimcpy  
despatch\_signals  
die  
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\_all  
dump\_eval  
dump\_fds  
dump\_form  
dump\_indent  
dump\_mstats  
dump\_packsubs  
dump\_sub  
dump\_vindent  
fetch\_cop\_label  
filter\_add  
filter\_del  
filter\_read  
find\_rundefsvoffset  
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\_check  
gv\_dump  
gv\_efullname  
gv\_efullname3  
gv\_efullname4



gv\_fetchfile  
gv\_fetchfile\_flags  
gv\_fetchmethod\_flags  
gv\_fetchpv  
gv\_fetchpvn\_flags  
gv\_fetchsv  
gv\_fullname  
gv\_fullname3  
gv\_fullname4  
gv\_handler  
gv\_init  
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\_p  
hv\_placeholders\_set  
hv\_riter\_p  
hv\_riter\_set  
hv\_store\_flags  
ibcmp  
ibcmp\_locale  
init\_global\_struct  
init\_j18n10n  
init\_j18n14n  
init\_stacks  
init\_tm  
instr  
is\_lvalue\_sub  
is\_uni\_alnum  
is\_uni\_alnum\_lc  
is\_uni\_alpha  
is\_uni\_alpha\_lc  
is\_uni\_ascii

is\_uni\_ascii\_lc  
is\_uni\_cntrl  
is\_uni\_cntrl\_lc  
is\_uni\_digit  
is\_uni\_digit\_lc  
is\_uni\_graph  
is\_uni\_graph\_lc  
is\_uni\_idfirst  
is\_uni\_idfirst\_lc  
is\_uni\_lower  
is\_uni\_lower\_lc  
is\_uni\_print  
is\_uni\_print\_lc  
is\_uni\_punct  
is\_uni\_punct\_lc  
is\_uni\_space  
is\_uni\_space\_lc  
is\_uni\_upper  
is\_uni\_upper\_lc  
is\_uni\_xdigit  
is\_uni\_xdigit\_lc  
is\_utf8\_alnum  
is\_utf8\_alpha  
is\_utf8\_ascii  
is\_utf8\_cntrl  
is\_utf8\_digit  
is\_utf8\_graph  
is\_utf8\_idcont  
is\_utf8\_idfirst  
is\_utf8\_lower  
is\_utf8\_mark  
is\_utf8\_perl\_space  
is\_utf8\_perl\_word  
is\_utf8\_posix\_digit  
is\_utf8\_print  
is\_utf8\_punct  
is\_utf8\_space  
is\_utf8\_upper  
is\_utf8\_xdigit  
leave\_scope  
load\_module\_nocontext  
magic\_dump

malloc  
markstack\_grow  
mess  
mess\_nocontext  
mfree  
mg\_dup  
mg\_size  
mini\_mktime  
moreswitches  
mro\_get\_from\_name  
mro\_get\_private\_data  
mro\_register  
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\_htonl  
my\_lstat  
my\_memcmp  
my\_memset  
my\_ntohl  
my\_pclose  
my\_popen  
my\_popen\_list  
my\_setenv  
my\_socketpair  
my\_stat  
my\_strftime  
my\_strlcat  
my\_strlcpy  
my\_swap  
newANONATTRSUB

newANONHASH  
newANONLIST  
newANONSUB  
newASSIGNOP  
newATTRSUB  
newAVREF  
newBINOP  
newCONDOP  
newCVREF  
newFORM  
newFOROP  
newGIVENOP  
newGVOP  
newGVREF  
newGVgen  
newHVREF  
newHVhv  
newIO  
newLISTOP  
newLOGOP  
newLOOPEX  
newLOOPOP  
newMYSUB  
newNULLLIST  
newOP  
newPADOP  
newPMOP  
newPROG  
newPVOP  
newRANGE  
newRV  
newSLICEOP  
newSTATEOP  
newSUB  
newSVOP  
newSVREF  
newSVpvf\_nocontext  
newUNOP  
newWHENOP  
newWHILEOP  
newXS\_flags  
new\_collate

new\_ctype  
new\_numeric  
new\_stackinfo  
ninstr  
op\_dump  
op\_free  
op\_null  
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\_clear  
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

regclass\_swash  
regdump  
regdupe\_internal  
regexec\_flags  
regfree\_internal  
reginitcolors  
regnext  
repeatcpy  
rninstr  
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\_freepv  
save\_freesv  
save\_generic\_pvref  
save\_generic\_svref  
save\_gp  
save\_hash  
save\_hdelete  
save\_helem  
save\_helem\_flags  
save\_hpتر

save\_int  
save\_item  
save\_iv  
save\_list  
save\_long  
save\_mortalizesv  
save\_nogv  
save\_padsv\_and\_mortalize  
save\_pptr  
save\_pushptr  
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  
screaminstr  
seed  
set\_context  
set\_numeric\_local  
set\_numeric\_radix  
set\_numeric\_standard  
share\_hek  
si\_dup  
ss\_dup  
stack\_grow  
start\_subparse  
stashpv\_hvname\_match  
str\_to\_version  
sv\_2iv  
sv\_2pv  
sv\_2uv  
sv\_catpvf\_mg\_nocontext  
sv\_catpvf\_nocontext  
sv\_compile\_2op  
sv\_dump  
sv\_dup

sv\_peek  
sv\_pvn\_nomg  
sv\_setpvf\_mg\_nocontext  
sv\_setpvf\_nocontext  
sv\_utf8\_upgrade\_flags\_grow  
swash\_fetch  
swash\_init  
sys\_init  
sys\_init3  
sys\_intern\_clear  
sys\_intern\_dup  
sys\_intern\_init  
sys\_term  
taint\_env  
taint\_proper  
tmpr\_grow  
to\_uni\_fold  
to\_uni\_lower  
to\_uni\_lower\_lc  
to\_uni\_title  
to\_uni\_title\_lc  
to\_uni\_upper  
to\_uni\_upper\_lc  
unlnk  
unsharepvn  
utf16\_to\_utf8  
utf16\_to\_utf8\_reversed  
uvchr\_to\_utf8\_flags  
uvuni\_to\_utf8  
vcroak  
vdeb  
vform  
vload\_module  
vmess  
vnewSVpvf  
vwarn  
vwarner  
warn\_nocontext  
warner  
warner\_nocontext  
whichsig



**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, perlxs, perlxsstut, perlintern*