

NAME

perlintern - autogenerated documentation of purely internal Perl functions

DESCRIPTION

This file is the autogenerated documentation of functions in the Perl interpreter that are documented using Perl's internal documentation format but are not marked as part of the Perl API. In other words, **they are not for use in extensions!**

It has the same sections as perlapi, though some may be empty.

AV Handling

"av_fetch_simple"

This is a cut-down version of av_fetch that assumes that the array is very straightforward - no magic, not readonly, and AvREAL - and that "key" is not negative. This function **MUST NOT** be used in situations where any of those assumptions may not hold.

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

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

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

"AvFILLp"

If the array "av" is empty, this returns -1; otherwise it returns the maximum value of the indices of all the array elements which are currently defined in "av". It does not handle magic, hence the "p" private indication in its name.

```
SSize_t AvFILLp(AV* av)
```

"av_new_alloc"

This implements ""newAV_alloc_x"" in perlapi and ""newAV_alloc_xz"" in perlapi, which are the public API for this functionality.

Creates a new AV and allocates its SV* array.

This is similar to, but more efficient than doing:

```
AV *av = newAV();
```

```
av_extend(av, key);
```

The size parameter is used to pre-allocate a SV* array large enough to hold at least elements "0..(size-1)". "size" must be at least 1.

The "zeroflag" parameter controls whether or not the array is NULL initialized.

```
AV* av_new_alloc(SSize_t size, bool zeroflag)
```

"av_store_simple"

This is a cut-down version of av_store that assumes that the array is very straightforward - no magic, not readonly, and AvREAL - and that "key" is not negative. This function **MUST NOT** be used in situations where any of those assumptions may not hold.

Stores an SV in an array. The array index is specified as "key". It can be dereferenced to get the "SV*" that was stored there (= "val").

Note that the caller is responsible for suitably incrementing the reference count of "val" before the call.

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

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

Callback Functions

"dowantarray"

Implements the deprecated ""GIMME"" in perlapi.

```
U8 dowantarray()
```

"leave_scope"

Implements "LEAVE_SCOPE" which you should use instead.

```
void leave_scope(I32 base)
```

"pop_scope"

Implements ""LEAVE"" in perlapi

```
void pop_scope()
```

"push_scope"

Implements ""ENTER"" in perlapi

void push_scope()

"save_adelete"

Implements "SAVEADELETE".

void save_adelete(AV *av, SSize_t key)

"save_generic_pvref"

Implements "SAVEGENERICPV".

Like **save_pptr()**, but also **Safefree()**s the new value if it is different from the old one. Can be used to restore a global char* to its prior contents, freeing new value.

void save_generic_pvref(char** str)

"save_generic_svref"

Implements "SAVEGENERICSV".

Like **save_sptr()**, but also **SvREFCNT_dec()**s the new value. Can be used to restore a global SV to its prior contents, freeing new value.

void save_generic_svref(SV** sptr)

"save_hdelete"

Implements "SAVEHDELETE".

void save_hdelete(HV *hv, SV *keysv)

"save_hints"

Implements "SAVEHINTS".

void save_hints()

"save_op"

Implements "SAVEOP".

void save_op()

"save_padsv_and_mortalize"

Implements "SAVEPADSVANDMORTALIZE".

```
void save_padsv_and_mortalize(PADOFFSET off)
```

"save_set_svflags"

Implements "SAVESETSVFLAGS".

Set the SvFLAGS specified by mask to the values in val

```
void save_set_svflags(SV *sv, U32 mask, U32 val)
```

"save_shared_pvref"

Implements "SAVESHAREDPV".

Like **save_generic_pvref()**, but uses **PerlMemShared_free()** rather than **Safefree()**. Can be used to restore a shared global char* to its prior contents, freeing new value.

```
void save_shared_pvref(char** str)
```

"save_vptr"

Implements "SAVEVPTR".

```
void save_vptr(void *ptr)
```

Casting

There are only public API items currently in Casting

Character case changing

There are only public API items currently in Character case changing

Character classification

There are only public API items currently in Character classification

Compiler and Preprocessor information

There are only public API items currently in Compiler and Preprocessor information

Compiler directives

There are only public API items currently in Compiler directives

Compile-time scope hooks**"BhkENTRY"**

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

Return an entry from the BHK structure. "which" is a preprocessor token indicating which entry to return. If the appropriate flag is not set this will return "NULL". The type of the return value depends on which entry you ask for.

```
void * BhkENTRY(BHK *hk, which)
```

"BhkFLAGS"

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

Return the BHK's flags.

```
U32 BhkFLAGS(BHK *hk)
```

"CALL_BLOCK_HOOKS"

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

Call all the registered block hooks for type "which". "which" is a preprocessing token; the type of "arg" depends on "which".

```
void CALL_BLOCK_HOOKS(which, arg)
```

Concurrency**"CVf_SLABBED"****"CvROOT"****"CvSTART"**

Described in perlguits.

"CX_CUR"

Described in perlguits.

```
CX_CUR()
```

"CXINC"

Described in perlguits.

"CX_LEAVE_SCOPE"

Described in perlguits.

```
void CX_LEAVE_SCOPE(PERL_CONTEXT* cx)
```

"CX_POP"

Described in perlguits.

```
void CX_POP(PERL_CONTEXT* cx)
```

"cxstack"

Described in perlguits.

"cxstack_ix"

Described in perlguits.

"CXt_BLOCK"

"CXt_EVAL"

"CXt_FORMAT"

"CXt_GIVEN"

"CXt_LOOP_ARY"

"CXt_LOOP_LAZYIV"

"CXt_LOOP_LAZYSV"

"CXt_LOOP_LIST"

"CXt_LOOP_PLAIN"

"CXt_NULL"

"CXt_SUB"

"CXt_SUBST"

"CXt_WHEN"

Described in perlguits.

"cx_type"

Described in perlguits.

"dounwind"

Described in perlguits.

```
void dounwind(I32 cxix)
```

"my_fork"

This is for the use of "PerlProc_fork" as a wrapper for the C library **fork**(2) on some platforms to hide some platform quirks. It should not be used except through "PerlProc_fork".

```
Pid_t my_fork()
```

```
"PERL_CONTEXT"
```

Described in perl guts.

COPs and Hint Hashes

There are only public API items currently in COPs and Hint Hashes

Custom Operators

```
"core_prototype"
```

This function assigns the prototype of the named core function to "sv", or to a new mortal SV if "sv" is "NULL". It returns the modified "sv", or "NULL" if the core function has no prototype. "code" is a code as returned by "keyword()". It must not be equal to 0.

```
SV * core_prototype(SV *sv, const char *name, const int code,
                    int * const opnum)
```

CV Handling

```
"CvWEAKOUTSIDE"
```

Each CV has a pointer, "CvOUTSIDE()", to its lexically enclosing CV (if any). Because pointers to anonymous sub prototypes are stored in "&" pad slots, it is possible to get a circular reference, with the parent pointing to the child and vice-versa. To avoid the ensuing memory leak, we do not increment the reference count of the CV pointed to by "CvOUTSIDE" in the *one specific instance* that the parent has a "&" pad slot pointing back to us. In this case, we set the "CvWEAKOUTSIDE" flag in the child. This allows us to determine under what circumstances we should decrement the refcount of the parent when freeing the child.

There is a further complication with non-closure anonymous subs (i.e. those that do not refer to any lexicals outside that sub). In this case, the anonymous prototype is shared rather than being cloned. This has the consequence that the parent may be freed while there are still active children, *e.g.*,

```
BEGIN { $a = sub { eval '$x' } }
```

In this case, the BEGIN is freed immediately after execution since there are no active references to it: the anon sub prototype has "CvWEAKOUTSIDE" set since it's not a closure, and \$a points to the same CV, so it doesn't contribute to BEGIN's refcount either. When \$a is executed, the "eval

'\$x' causes the chain of "CvOUTSIDE"s to be followed, and the freed BEGIN is accessed.

To avoid this, whenever a CV and its associated pad is freed, any "&" entries in the pad are explicitly removed from the pad, and if the refcount of the pointed-to anon sub is still positive, then that child's "CvOUTSIDE" is set to point to its grandparent. This will only occur in the single specific case of a non-closure anon prototype having one or more active references (such as \$a above).

One other thing to consider is that a CV may be merely undefined rather than freed, eg "undef &foo". In this case, its refcount may not have reached zero, but we still delete its pad and its "CvROOT" etc. Since various children may still have their "CvOUTSIDE" pointing at this undefined CV, we keep its own "CvOUTSIDE" for the time being, so that the chain of lexical scopes is unbroken. For example, the following should print 123:

```
my $x = 123;
sub tmp { sub { eval '$x' } }
my $a = tmp();
undef &tmp;
print $a->();
```

```
bool CvWEAKOUTSIDE(CV *cv)
```

"docatch"

Check for the cases 0 or 3 of cur_env.je_ret, only used inside an eval context.

0 is used as continue inside eval,

3 is used for a die caught by an inner eval - continue inner loop

See *cop.h*: je_mustcatch, when set at any runlevel to TRUE, means eval ops must establish a local jmpenv to handle exception traps.

```
OP* docatch(Perl_ppaddr_t firstpp)
```

Debugging

"_aDEPTH"

Some functions when compiled under DEBUGGING take an extra final argument named "depth", indicating the C stack depth. This argument is omitted otherwise. This macro expands to either ", depth" under DEBUGGING, or to nothing at all when not under DEBUGGING, reducing the number of "#ifdef"s in the code.

The program is responsible for maintaining the correct value for "depth".

`_aDEPTH`

"debop"

Implements **-Dt** perl command line option on OP "o".

I32 debop(const OP* o)

"debprof"

Called to indicate that "o" was executed, for profiling purposes under the "-DP" command line option.

void debprof(const OP *o)

"debprofdump"

Dumps the contents of the data collected by the "-DP" perl command line option.

void debprofdump()

"free_c_backtrace"

Deallocates a backtrace received from get_c_backtrace.

void free_c_backtrace(Perl_c_backtrace* bt)

"get_c_backtrace"

Collects the backtrace (aka "stacktrace") into a single linear malloced buffer, which the caller **must** "Perl_free_c_backtrace()".

Scans the frames back by "depth + skip", then drops the "skip" innermost, returning at most "depth" frames.

Perl_c_backtrace* get_c_backtrace(int max_depth, int skip)

"_pDEPTH"

This is used in the prototype declarations for functions that take a ""_aDEPTH"" final parameter, much like "pTHX_" is used in functions that take a thread context initial parameter.

"PL_DBsingle"

When Perl is run in debugging mode, with the **-d** switch, this SV is a boolean which indicates

whether subs are being single-stepped. Single-stepping is automatically turned on after every step. This is the C variable which corresponds to Perl's \$DB::single variable. See "PL_DBsub".

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

SV * PL_DBsingle

"PL_DBsub"

When Perl is run in debugging mode, with the **-d** switch, this GV contains the SV which holds the name of the sub being debugged. This is the C variable which corresponds to Perl's \$DB::sub variable. See "PL_DBsingle".

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

GV * PL_DBsub

"PL_DBtrace"

Trace variable used when Perl is run in debugging mode, with the **-d** switch. This is the C variable which corresponds to Perl's \$DB::trace variable. See "PL_DBsingle".

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

SV * PL_DBtrace

"runops_debug"

Described in perl guts.

int runops_debug()

"runops_standard"

Described in perl guts.

int runops_standard()

Display functions

"sv_peek"

Implements "SvPEEK"

```
char* sv_peek(SV* sv)
```

Embedding, Threads, and Interpreter Cloning

"cv_dump"

dump the contents of a CV

```
void cv_dump(const CV *cv, const char *title)
```

"cv_forget_slab"

When a CV has a reference count on its slab ("CvSLABBED"), it is responsible for making sure it is freed. (Hence, no two CVs should ever have a reference count on the same slab.) The CV only needs to reference the slab during compilation. Once it is compiled and "CvROOT" attached, it has finished its job, so it can forget the slab.

```
void cv_forget_slab(CV *cv)
```

"do_dump_pad"

Dump the contents of a padlist

```
void do_dump_pad(I32 level, PerlIO *file, PADLIST *padlist,
                int full)
```

"get_context"

Implements ""PERL_GET_CONTEXT"" in perlapi, which you should use instead.

```
void* get_context()
```

"pad_alloc_name"

Allocates a place in the currently-compiling pad (via "pad_alloc" in perlapi) and then stores a name for that entry. "name" is adopted and becomes the name entry; it must already contain the name string. "typestash" and "ourstash" and the "padadd_STATE" flag get added to "name". None of the other processing of "pad_add_name_pvn" in perlapi is done. Returns the offset of the allocated pad slot.

```
PADOFFSET pad_alloc_name(PADNAME *name, U32 flags, HV *typestash,
                        HV *ourstash)
```

"pad_block_start"

Update the pad compilation state variables on entry to a new block.

```
void pad_block_start(int full)
```

"pad_check_dup"

Check for duplicate declarations: report any of:

- * a 'my' in the current scope with the same name;
- * an 'our' (anywhere in the pad) with the same name and the same stash as 'ourstash'

"is_our" indicates that the name to check is an "our" declaration.

```
void pad_check_dup(PADNAME *name, U32 flags, const HV *ourstash)
```

"pad_findlex"

Find a named lexical anywhere in a chain of nested pads. Add fake entries in the inner pads if it's found in an outer one.

Returns the offset in the bottom pad of the lex or the fake lex. "cv" is the CV in which to start the search, and seq is the current "cop_seq" to match against. If "warn" is true, print appropriate warnings. The "out_*" vars return values, and so are pointers to where the returned values should be stored. "out_capture", if non-null, requests that the innermost instance of the lexical is captured; "out_name" is set to the innermost matched pad name or fake pad name; "out_flags" returns the flags normally associated with the "PARENT_FAKELEX_FLAGS" field of a fake pad name.

Note that "pad_findlex()" is recursive; it recurses up the chain of CVs, then comes back down, adding fake entries as it goes. It has to be this way because fake names in anon prototypes have to store in "xpadn_low" the index into the parent pad.

```
PADOFFSET pad_findlex(const char *namepv, STRLEN namelen,
                      U32 flags, const CV* cv, U32 seq, int warn,
                      SV** out_capture, PADNAME** out_name,
                      int *out_flags)
```

"pad_fixup_inner_anons"

For any anon CVs in the pad, change "CvOUTSIDE" of that CV from "old_cv" to "new_cv" if necessary. Needed when a newly-compiled CV has to be moved to a pre-existing CV struct.

```
void pad_fixup_inner_anons(PADLIST *padlist, CV *old_cv,
                          CV *new_cv)
```

`"pad_free"`

Free the SV at offset `po` in the current pad.

```
void pad_free(PADOFFSET po)
```

`"pad_leavemy"`

Cleanup at end of scope during compilation: set the max seq number for lexicals in this scope and warn of any lexicals that never got introduced.

```
OP * pad_leavemy()
```

`"padlist_dup"`

Duplicates a pad.

```
PADLIST * padlist_dup(PADLIST *srcpad, CLONE_PARAMS *param)
```

`"padname_dup"`

Duplicates a pad name.

```
PADNAME * padname_dup(PADNAME *src, CLONE_PARAMS *param)
```

`"padnamelist_dup"`

Duplicates a pad name list.

```
PADNAMELIST * padnamelist_dup(PADNAMELIST *srcpad,  
                               CLONE_PARAMS *param)
```

`"pad_push"`

Push a new pad frame onto the padlist, unless there's already a pad at this depth, in which case don't bother creating a new one. Then give the new pad an `@_` in slot zero.

```
void pad_push(PADLIST *padlist, int depth)
```

`"pad_reset"`

Mark all the current temporaries for reuse

```
void pad_reset()
```

`"pad_setsv"`

Set the value at offset `"po"` in the current (compiling or executing) pad. Use the macro

"PAD_SETSV()" rather than calling this function directly.

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

"pad_sv"

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

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

"pad_swipe"

Abandon the tmp in the current pad at offset "po" and replace with a new one.

```
void pad_swipe(PADOFFSET po, bool refadjust)
```

"set_context"

Implements ""PERL_SET_CONTEXT"" in perlapi, which you should use instead.

```
void set_context(void *t)
```

Errno

"dSAVEDERRNO"

Declare variables needed to save "errno" and any operating system specific error number.

```
void dSAVEDERRNO
```

"dSAVE_ERRNO"

Declare variables needed to save "errno" and any operating system specific error number, and save them for optional later restoration by "RESTORE_ERRNO".

```
void dSAVE_ERRNO
```

"RESTORE_ERRNO"

Restore "errno" and any operating system specific error number that was saved by "dSAVE_ERRNO" or "RESTORE_ERRNO".

```
void RESTORE_ERRNO
```

"SAVE_ERRNO"

Save "errno" and any operating system specific error number for optional later restoration by

"RESTORE_ERRNO". Requires "dSAVEDERRNO" or "dSAVE_ERRNO" in scope.

```
void SAVE_ERRNO
```

"SETERRNO"

Set "errno", and on VMS set "vaxc\$errno".

```
void SETERRNO(int errcode, int vmserrcode)
```

Exception Handling (simple) Macros

There are only public API items currently in Exception Handling (simple) Macros

Filesystem configuration values

There are only public API items currently in Filesystem configuration values

Floating point

There are only public API items currently in Floating point

General Configuration

There are only public API items currently in General Configuration

Global Variables

There are only public API items currently in Global Variables

GV Handling and Stashes

"gp_dup"

Duplicate a typeglob, returning a pointer to the cloned object.

```
GP* gp_dup(GP *const gp, CLONE_PARAMS *const param)
```

"gv_handler"

Implements "StashHANDLER", which you should use instead

```
CV* gv_handler(HV* stash, I32 id)
```

"gv_stashvpvn_cached"

Returns a pointer to the stash for a specified package, possibly cached. Implements both ""gv_stashpvn"" in perlapi and ""gv_stashsv"" in perlapi.

Requires one of either "namesv" or "namepv" to be non-null.

If the flag "GV_CACHE_ONLY" is set, return the stash only if found in the cache; see ""gv_stashpv" in perlapi for details on the other "flags".

Note it is strongly preferred for "namesv" to be non-null, for performance reasons.

```
HV* gv_stashpv_cached(SV *namesv, const char* name,
                    U32 namelen, I32 flags)
```

"gv_try_downgrade"

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

If the typeglob "gv" can be expressed more succinctly, by having something other than a real GV in its place in the stash, replace it with the optimised form. Basic requirements for this are that "gv" is a real typeglob, is sufficiently ordinary, and is only referenced from its package. This function is meant to be used when a GV has been looked up in part to see what was there, causing upgrading, but based on what was found it turns out that the real GV isn't required after all.

If "gv" is a completely empty typeglob, it is deleted from the stash.

If "gv" is a typeglob containing only a sufficiently-ordinary constant sub, the typeglob is replaced with a scalar-reference placeholder that more compactly represents the same thing.

```
void gv_try_downgrade(GV* gv)
```

Hook manipulation

There are only public API items currently in Hook manipulation

HV Handling

"hv_eiter_p"

Implements "HvEITER" which you should use instead.

NOTE: "hv_eiter_p" must be explicitly called as "Perl_hv_eiter_p" with an "aTHX_" parameter.

```
HE** Perl_hv_eiter_p(pTHX_ HV *hv)
```

"hv_eiter_set"

Implements "HvEITER_set" which you should use instead.

NOTE: "hv_eiter_set" must be explicitly called as "Perl_hv_eiter_set" with an "aTHX_" parameter.


```
void Perl_hv_eiter_set(pTHX_ HV *hv, HE *eiter)
```

"hv_ename_add"

Adds a name to a stash's internal list of effective names. See "hv_ename_delete".

This is called when a stash is assigned to a new location in the symbol table.

```
void hv_ename_add(HV *hv, const char *name, U32 len, U32 flags)
```

"hv_ename_delete"

Removes a name from a stash's internal list of effective names. If this is the name returned by "HvENAME", then another name in the list will take its place ("HvENAME" will use it).

This is called when a stash is deleted from the symbol table.

```
void hv_ename_delete(HV *hv, const char *name, U32 len,
                    U32 flags)
```

"hv_fill"

Returns the number of hash buckets that happen to be in use.

This function implements the "HvFILL" macro which you should use instead.

As of perl 5.25 this function is used only for debugging purposes, and the number of used hash buckets is not in any way cached, thus this function can be costly to execute as it must iterate over all the buckets in the hash.

NOTE: "hv_fill" must be explicitly called as "Perl_hv_fill" with an "aTHX_" parameter.

```
STRLEN Perl_hv_fill(pTHX_ HV *const hv)
```

"hv_placeholders_get"

Implements "HvPLACEHOLDERS_get", which you should use instead.

NOTE: "hv_placeholders_get" must be explicitly called as "Perl_hv_placeholders_get" with an "aTHX_" parameter.

```
I32 Perl_hv_placeholders_get(pTHX_ const HV *hv)
```

"hv_placeholders_set"

Implements "HvPLACEHOLDERS_set", which you should use instead.

NOTE: "hv_placeholders_set" must be explicitly called as "Perl_hv_placeholders_set" with an "aTHX_" parameter.

```
void Perl_hv_placeholders_set(pTHX_ HV *hv, I32 ph)
```

"hv_riter_p"

Implements "HvRITER" which you should use instead.

NOTE: "hv_riter_p" must be explicitly called as "Perl_hv_riter_p" with an "aTHX_" parameter.

```
I32* Perl_hv_riter_p(pTHX_ HV *hv)
```

"hv_riter_set"

Implements "HvRITER_set" which you should use instead.

NOTE: "hv_riter_set" must be explicitly called as "Perl_hv_riter_set" with an "aTHX_" parameter.

```
void Perl_hv_riter_set(pTHX_ HV *hv, I32 riter)
```

"refcounted_he_chain_2hv"

Generates and returns a "HV *" representing the content of a "refcounted_he" chain. "flags" is currently unused and must be zero.

```
HV * refcounted_he_chain_2hv(const struct refcounted_he *c,
                             U32 flags)
```

"refcounted_he_fetch_pv"

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

```
SV * refcounted_he_fetch_pv(const struct refcounted_he *chain,
                             const char *key, U32 hash, U32 flags)
```

"refcounted_he_fetch_pvn"

Search along a "refcounted_he" chain for an entry with the key specified by "keypv" and "keylen". If "flags" has the "REFCOUNTED_HE_KEY_UTF8" bit set, the key octets are interpreted as UTF-8, otherwise they are interpreted as Latin-1. "hash" is a precomputed hash of the key string, or zero if it has not been precomputed. Returns a mortal scalar representing the value associated

with the key, or `&PL_sv_placeholder` if there is no value associated with the key.

```
SV * refcounted_he_fetch_pvn(const struct refcounted_he *chain,
                             const char *keypv, STRLEN keylen,
                             U32 hash, U32 flags)
```

"refcounted_he_fetch_pvs"

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

```
SV * refcounted_he_fetch_pvs(const struct refcounted_he *chain,
                             "key", U32 flags)
```

"refcounted_he_fetch_sv"

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

```
SV * refcounted_he_fetch_sv(const struct refcounted_he *chain,
                             SV *key, U32 hash, U32 flags)
```

"refcounted_he_free"

Decrements the reference count of a "refcounted_he" by one. If the reference count reaches zero the structure's memory is freed, which (recursively) causes a reduction of its parent "refcounted_he"'s reference count. It is safe to pass a null pointer to this function: no action occurs in this case.

```
void refcounted_he_free(struct refcounted_he *he)
```

"refcounted_he_inc"

Increment the reference count of a "refcounted_he". The pointer to the "refcounted_he" is also returned. It is safe to pass a null pointer to this function: no action occurs and a null pointer is returned.

```
struct refcounted_he * refcounted_he_inc(
    struct refcounted_he *he)
```

"refcounted_he_new_pvn"

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

```
struct refcounted_he * refcounted_he_new_pvn(
    struct refcounted_he *parent,
```

```
const char *key, U32 hash,
SV *value, U32 flags)
```

"refcounted_he_new_pvn"

Creates a new "refcounted_he". This consists of a single key/value pair and a reference to an existing "refcounted_he" chain (which may be empty), and thus forms a longer chain. When using the longer chain, the new key/value pair takes precedence over any entry for the same key further along the chain.

The new key is specified by "keypv" and "keylen". If "flags" has the "REFCOUNTED_HE_KEY_UTF8" bit set, the key octets are interpreted as UTF-8, otherwise they are interpreted as Latin-1. "hash" is a precomputed hash of the key string, or zero if it has not been precomputed.

"value" is the scalar value to store for this key. "value" is copied by this function, which thus does not take ownership of any reference to it, and later changes to the scalar will not be reflected in the value visible in the "refcounted_he". Complex types of scalar will not be stored with referential integrity, but will be coerced to strings. "value" may be either null or &PL_sv_placeholder to indicate that no value is to be associated with the key; this, as with any non-null value, takes precedence over the existence of a value for the key further along the chain.

"parent" points to the rest of the "refcounted_he" chain to be attached to the new "refcounted_he". This function takes ownership of one reference to "parent", and returns one reference to the new "refcounted_he".

```
struct refcounted_he * refcounted_he_new_pvn(
    struct refcounted_he *parent,
    const char *keypv,
    STRLEN keylen, U32 hash,
    SV *value, U32 flags)
```

"refcounted_he_new_pvs"

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

```
struct refcounted_he * refcounted_he_new_pvs(
    struct refcounted_he *parent,
    "key", SV *value, U32 flags)
```

"refcounted_he_new_sv"

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

```
struct refcounted_he * refcounted_he_new_sv(
    struct refcounted_he *parent,
    SV *key, U32 hash, SV *value,
    U32 flags)
```

"unsharepvn"

If no one has access to shared string "str" with length "len", free it.

"len" and "hash" must both be valid for "str".

```
void unsharepvn(const char* sv, I32 len, U32 hash)
```

Input/Output

"dirp_dup"

Duplicate a directory handle, returning a pointer to the cloned object.

```
DIR* dirp_dup(DIR *const dp, CLONE_PARAMS *const param)
```

"fp_dup"

Duplicate a file handle, returning a pointer to the cloned object.

```
PerlIO* fp_dup(PerlIO *const fp, const char type,
    CLONE_PARAMS *const param)
```

"my_fflush_all"

Implements "PERL_FLUSHALL_FOR_CHILD" on some platforms.

```
I32 my_fflush_all()
```

"my_mkostemp"

The C library mkostemp(3) if available, or a Perl implementation of it.

NOTE: "my_mkostemp" must be explicitly called as "Perl_my_mkostemp" .

```
int Perl_my_mkostemp(char *templte, int flags)
```

"my_mkstemp"

The C library mkstemp(3) if available, or a Perl implementation of it.

NOTE: "my_mkstemp" must be explicitly called as "Perl_my_mkstemp" .

```
int Perl_my_mkstemp(char *templte)
```

"PL_last_in_gv"

The GV which was last used for a filehandle input operation. ("`<FH>`")

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

```
GV* PL_last_in_gv
```

"PL_ofsgv"

The glob containing the output field separator - "`*`," in Perl space.

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

```
GV* PL_ofsgv
```

"PL_rs"

The input record separator - `$/` in Perl space.

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

```
SV* PL_rs
```

"start_glob"

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

Function called by "do_readline" to spawn a glob (or do the glob inside perl on VMS). This code used to be inline, but now perl uses "File::Glob" this glob starter is only used by miniperl during the build process, or when PERL_EXTERNAL_GLOB is defined. Moving it away shrinks *pp_hot.c*; shrinking *pp_hot.c* helps speed perl up.

NOTE: "start_glob" must be explicitly called as "Perl_start_glob" with an "aTHX_" parameter.

```
PerlIO* Perl_start_glob(pTHX_ SV *tmpglob, IO *io)
```

Integer

There are only public API items currently in Integer

I/O Formats

There are only public API items currently in I/O Formats

Lexer interface

"validate_proto"

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

This function performs syntax checking on a prototype, "proto". If "warn" is true, any illegal characters or mismatched brackets will trigger illegalproto warnings, declaring that they were detected in the prototype for "name".

The return value is "true" if this is a valid prototype, and "false" if it is not, regardless of whether "warn" was "true" or "false".

Note that "NULL" is a valid "proto" and will always return "true".

```
bool validate_proto(SV *name, SV *proto, bool warn,  
                  bool curstash)
```

Locales

There are only public API items currently in Locales

Magic

"magic_clearhint"

Triggered by a delete from "%^H", records the key to "PL_compiling.cop_hints_hash".

```
int magic_clearhint(SV* sv, MAGIC* mg)
```

"magic_clearhints"

Triggered by clearing "%^H", resets "PL_compiling.cop_hints_hash".

```
int magic_clearhints(SV* sv, MAGIC* mg)
```

"magic_methcall"

Invoke a magic method (like FETCH).

"sv" and "mg" are the tied thingy and the tie magic.

"meth" is the name of the method to call.

"argc" is the number of args (in addition to \$self) to pass to the method.

The "flags" can be:

```
G_DISCARD  invoke method with G_DISCARD flag and don't
             return a value
G_UNDEF_FILL fill the stack with argc pointers to
             PL_sv_undef
```

The arguments themselves are any values following the "flags" argument.

Returns the SV (if any) returned by the method, or "NULL" on failure.

NOTE: "magic_methcall" must be explicitly called as "Perl_magic_methcall" with an "aTHX_" parameter.

```
SV* Perl_magic_methcall(pTHX_ SV *sv, const MAGIC *mg, SV *meth,
                       U32 flags, U32 argc, ...)
```

"magic_sethint"

Triggered by a store to "%^H", records the key/value pair to "PL_compiling.cop_hints_hash". It is assumed that hints aren't storing anything that would need a deep copy. Maybe we should warn if we find a reference.

```
int magic_sethint(SV* sv, MAGIC* mg)
```

"mg_dup"

Duplicate a chain of magic, returning a pointer to the cloned object.

```
MAGIC* mg_dup(MAGIC *mg, CLONE_PARAMS *const param)
```

"mg_localize"

Copy some of the magic from an existing SV to new localized version of that SV. Container magic (e.g., %ENV, \$1, "tie") gets copied, value magic doesn't (e.g., "taint", "pos").

If "setmagic" is false then no set magic will be called on the new (empty) SV. This typically means that assignment will soon follow (e.g. 'local \$x = \$y'), and that will handle the magic.


```
void mg_localize(SV* sv, SV* nsv, bool setmagic)
```

"si_dup"

Duplicate a stack info structure, returning a pointer to the cloned object.

```
PERL_SI* si_dup(PERL_SI* si, CLONE_PARAMS* param)
```

"ss_dup"

Duplicate the save stack, returning a pointer to the cloned object.

```
ANY* ss_dup(PerlInterpreter* proto_perl, CLONE_PARAMS* param)
```

Memory Management

"calloc"

Implements ""Newxz"" in perlapi which you should use instead.

NOTE: "calloc" must be explicitly called as "Perl_calloc" .

```
Malloc_t Perl_calloc(MEM_SIZE elements, MEM_SIZE size)
```

"malloc"

Implements ""Newx"" in perlapi which you should use instead.

NOTE: "malloc" must be explicitly called as "Perl_malloc" .

```
Malloc_t Perl_malloc(MEM_SIZE nbytes)
```

"mfree"

Implements ""Safefree"" in perlapi which you should use instead.

NOTE: "mfree" must be explicitly called as "Perl_mfree" .

```
Free_t Perl_mfree(Malloc_t where)
```

"realloc"

Implements ""Renew"" in perlapi which you should use instead.

NOTE: "realloc" must be explicitly called as "Perl_realloc" .

```
Malloc_t Perl_realloc(Malloc_t where, MEM_SIZE nbytes)
```

MRO`"mro_get_linear_isa_dfs"`

Returns the Depth-First Search linearization of @ISA the given stash. The return value is a read-only AV*. "level" should be 0 (it is used internally in this function's recursion).

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_dfs(HV* stash, U32 level)
```

`"mro_isa_changed_in"`

Takes the necessary steps (cache invalidations, mostly) when the @ISA of the given package has changed. Invoked by the "setisa" magic, should not need to invoke directly.

```
void mro_isa_changed_in(HV* stash)
```

`"mro_package_moved"`

Call this function to signal to a stash that it has been assigned to another spot in the stash hierarchy. "stash" is the stash that has been assigned. "oldstash" is the stash it replaces, if any. "gv" is the glob that is actually being assigned to.

This can also be called with a null first argument to indicate that "oldstash" has been deleted.

This function invalidates isa caches on the old stash, on all subpackages nested inside it, and on the subclasses of all those, including non-existent packages that have corresponding entries in "stash".

It also sets the effective names ("HvENAME") on all the stashes as appropriate.

If the "gv" is present and is not in the symbol table, then this function simply returns. This checked will be skipped if "flags & 1".

```
void mro_package_moved(HV * const stash, HV * const oldstash,
                      const GV * const gv, U32 flags)
```

Multicall Functions

There are only public API items currently in Multicall Functions

Numeric Functions

"isinfnavsv"

Checks whether the argument would be either an infinity or "NaN" when used as a number, but is careful not to trigger non-numeric or uninitialized warnings. It assumes the caller has done "SvGETMAGIC(sv)" already.

Note that this always accepts trailing garbage (similar to "grok_number_flags" with "PERL_SCAN_TRAILING"), so "inferior" and "NAND gates" will return true.

```
bool isinfnavsv(SV *sv)
```

Optrees**"finalize_optree"**

This function finalizes the optree. Should be called directly after the complete optree is built. It does some additional checking which can't be done in the normal "ck_XXX" functions and makes the tree thread-safe.

```
void finalize_optree(OP* o)
```

"newATTRSUB_x"

Construct a Perl subroutine, also performing some surrounding jobs.

This function is expected to be called in a Perl compilation context, and some aspects of the subroutine are taken from global variables associated with compilation. In particular, "PL_compcv" represents the subroutine that is currently being compiled. It must be non-null when this function is called, and some aspects of the subroutine being constructed are taken from it. The constructed subroutine may actually be a reuse of the "PL_compcv" object, but will not necessarily be so.

If "block" is null then the subroutine will have no body, and for the time being it will be an error to call it. This represents a forward subroutine declaration such as "sub foo (\$\$);". If "block" is non-null then it provides the Perl code of the subroutine body, which will be executed when the subroutine is called. This body includes any argument unwrapping code resulting from a subroutine signature or similar. The pad use of the code must correspond to the pad attached to "PL_compcv". The code is not expected to include a "leavesub" or "leavesublv" op; this function will add such an op. "block" is consumed by this function and will become part of the constructed subroutine.

"proto" specifies the subroutine's prototype, unless one is supplied as an attribute (see below). If "proto" is null, then the subroutine will not have a prototype. If "proto" is non-null, it must point to a "const" op whose value is a string, and the subroutine will have that string as its prototype. If

a prototype is supplied as an attribute, the attribute takes precedence over "proto", but in that case "proto" should preferably be null. In any case, "proto" is consumed by this function.

"attrs" supplies attributes to be applied the subroutine. A handful of attributes take effect by built-in means, being applied to "PL_compcv" immediately when seen. Other attributes are collected up and attached to the subroutine by this route. "attrs" may be null to supply no attributes, or point to a "const" op for a single attribute, or point to a "list" op whose children apart from the "pushmark" are "const" ops for one or more attributes. Each "const" op must be a string, giving the attribute name optionally followed by parenthesised arguments, in the manner in which attributes appear in Perl source. The attributes will be applied to the sub by this function. "attrs" is consumed by this function.

If "o_is_gv" is false and "o" is null, then the subroutine will be anonymous. If "o_is_gv" is false and "o" is non-null, then "o" must point to a "const" OP, which will be consumed by this function, and its string value supplies a name for the subroutine. The name may be qualified or unqualified, and if it is unqualified then a default stash will be selected in some manner. If "o_is_gv" is true, then "o" doesn't point to an "OP" at all, but is instead a cast pointer to a "GV" by which the subroutine will be named.

If there is already a subroutine of the specified name, then the new sub will either replace the existing one in the glob or be merged with the existing one. A warning may be generated about redefinition.

If the subroutine has one of a few special names, such as "BEGIN" or "END", then it will be claimed by the appropriate queue for automatic running of phase-related subroutines. In this case the relevant glob will be left not containing any subroutine, even if it did contain one before. In the case of "BEGIN", the subroutine will be executed and the reference to it disposed of before this function returns.

The function returns a pointer to the constructed subroutine. If the sub is anonymous then ownership of one counted reference to the subroutine is transferred to the caller. If the sub is named then the caller does not get ownership of a reference. In most such cases, where the sub has a non-phase name, the sub will be alive at the point it is returned by virtue of being contained in the glob that names it. A phase-named subroutine will usually be alive by virtue of the reference owned by the phase's automatic run queue. But a "BEGIN" subroutine, having already been executed, will quite likely have been destroyed already by the time this function returns, making it erroneous for the caller to make any use of the returned pointer. It is the caller's responsibility to ensure that it knows which of these situations applies.

CV* newATTRSUB_x(I32 floor, OP *o, OP *proto, OP *attrs,

OP *block, bool o_is_gv)

"newXS_len_flags"

Construct an XS subroutine, also performing some surrounding jobs.

The subroutine will have the entry point "subaddr". It will have the prototype specified by the nul-terminated string "proto", or no prototype if "proto" is null. The prototype string is copied; the caller can mutate the supplied string afterwards. If "filename" is non-null, it must be a nul-terminated filename, and the subroutine will have its "CvFILE" set accordingly. By default "CvFILE" is set to point directly to the supplied string, which must be static. If "flags" has the "XS_DYNAMIC_FILENAME" bit set, then a copy of the string will be taken instead.

Other aspects of the subroutine will be left in their default state. If anything else needs to be done to the subroutine for it to function correctly, it is the caller's responsibility to do that after this function has constructed it. However, beware of the subroutine potentially being destroyed before this function returns, as described below.

If "name" is null then the subroutine will be anonymous, with its "CvGV" referring to an "__ANON__" glob. If "name" is non-null then the subroutine will be named accordingly, referenced by the appropriate glob. "name" is a string of length "len" bytes giving a sigilless symbol name, in UTF-8 if "flags" has the "SVf_UTF8" bit set and in Latin-1 otherwise. The name may be either qualified or unqualified, with the stash defaulting in the same manner as for "gv_fetchpvn_flags". "flags" may contain flag bits understood by "gv_fetchpvn_flags" with the same meaning as they have there, such as "GV_ADDWARN". The symbol is always added to the stash if necessary, with "GV_ADDMULTI" semantics.

If there is already a subroutine of the specified name, then the new sub will replace the existing one in the glob. A warning may be generated about the redefinition. If the old subroutine was "CvCONST" then the decision about whether to warn is influenced by an expectation about whether the new subroutine will become a constant of similar value. That expectation is determined by "const_svp". (Note that the call to this function doesn't make the new subroutine "CvCONST" in any case; that is left to the caller.) If "const_svp" is null then it indicates that the new subroutine will not become a constant. If "const_svp" is non-null then it indicates that the new subroutine will become a constant, and it points to an "SV*" that provides the constant value that the subroutine will have.

If the subroutine has one of a few special names, such as "BEGIN" or "END", then it will be claimed by the appropriate queue for automatic running of phase-related subroutines. In this case the relevant glob will be left not containing any subroutine, even if it did contain one before. In the case of "BEGIN", the subroutine will be executed and the reference to it disposed of before

this function returns, and also before its prototype is set. If a "BEGIN" subroutine would not be sufficiently constructed by this function to be ready for execution then the caller must prevent this happening by giving the subroutine a different name.

The function returns a pointer to the constructed subroutine. If the sub is anonymous then ownership of one counted reference to the subroutine is transferred to the caller. If the sub is named then the caller does not get ownership of a reference. In most such cases, where the sub has a non-phase name, the sub will be alive at the point it is returned by virtue of being contained in the glob that names it. A phase-named subroutine will usually be alive by virtue of the reference owned by the phase's automatic run queue. But a "BEGIN" subroutine, having already been executed, will quite likely have been destroyed already by the time this function returns, making it erroneous for the caller to make any use of the returned pointer. It is the caller's responsibility to ensure that it knows which of these situations applies.

```
CV * newXS_len_flags(const char *name, STRLEN len,
                    XSUBADDR_t subaddr,
                    const char *const filename,
                    const char *const proto, SV **const_svp,
                    U32 flags)
```

"op_refcnt_lock"

Implements the "OP_REFCNT_LOCK" macro which you should use instead.

```
void op_refcnt_lock()
```

"op_refcnt_unlock"

Implements the "OP_REFCNT_UNLOCK" macro which you should use instead.

```
void op_refcnt_unlock()
```

"optimize_optree"

This function applies some optimisations to the optree in top-down order. It is called before the peephole optimizer, which processes ops in execution order. Note that **finalize_optree()** also does a top-down scan, but is called *after* the peephole optimizer.

```
void optimize_optree(OP* o)
```

"traverse_op_tree"

Return the next op in a depth-first traversal of the op tree, returning NULL when the traversal is complete.

The initial call must supply the root of the tree as both top and o.

For now it's static, but it may be exposed to the API in the future.

```
OP* traverse_op_tree(OP* top, OP* o)
```

Pack and Unpack

There are only public API items currently in Pack and Unpack

Pad Data Structures

"CX_CURPAD_SAVE"

Save the current pad in the given context block structure.

```
void CX_CURPAD_SAVE(struct context)
```

"CX_CURPAD_SV"

Access the SV at offset "po" in the saved current pad in the given context block structure (can be used as an lvalue).

```
SV * CX_CURPAD_SV(struct context, PADOFFSET po)
```

"PAD_BASE_SV"

Get the value from slot "po" in the base (DEPTH=1) pad of a padlist

```
SV * PAD_BASE_SV(PADLIST padlist, PADOFFSET po)
```

"PAD_CLONE_VARS"

Clone the state variables associated with running and compiling pads.

```
void PAD_CLONE_VARS(PerlInterpreter *proto_perl,
                    CLONE_PARAMS* param)
```

"PAD_COMPNAME_FLAGS"

Return the flags for the current compiling pad name at offset "po". Assumes a valid slot entry.

```
U32 PAD_COMPNAME_FLAGS(PADOFFSET po)
```

"PAD_COMPNAME_GEN"

The generation number of the name at offset "po" in the current compiling pad (lvalue).

STRLEN PAD_COMPNAME_GEN(PADOFFSET po)

"PAD_COMPNAME_GEN_set"

Sets the generation number of the name at offset "po" in the current ling pad (lvalue) to "gen".

STRLEN PAD_COMPNAME_GEN_set(PADOFFSET po, int gen)

"PAD_COMPNAME_OURSTASH"

Return the stash associated with an "our" variable. Assumes the slot entry is a valid "our" lexical.

HV * PAD_COMPNAME_OURSTASH(PADOFFSET po)

"PAD_COMPNAME_PV"

Return the name of the current compiling pad name at offset "po". Assumes a valid slot entry.

char * PAD_COMPNAME_PV(PADOFFSET po)

"PAD_COMPNAME_TYPE"

Return the type (stash) of the current compiling pad name at offset "po". Must be a valid name.
Returns null if not typed.

HV * PAD_COMPNAME_TYPE(PADOFFSET po)

"PadnameIsOUR"

Whether this is an "our" variable.

bool PadnameIsOUR(PADNAME * pn)

"PadnameIsSTATE"

Whether this is a "state" variable.

bool PadnameIsSTATE(PADNAME * pn)

"PadnameOURSTASH"

The stash in which this "our" variable was declared.

HV * PadnameOURSTASH(PADNAME * pn)

"PadnameOUTER"

Whether this entry belongs to an outer pad. Entries for which this is true are often referred to as

'fake'.

bool PadnameOUTER(PADNAME * pn)

"PadnameTYPE"

The stash associated with a typed lexical. This returns the %Foo:: hash for "my Foo \$bar".

HV * PadnameTYPE(PADNAME * pn)

"PAD_RESTORE_LOCAL"

Restore the old pad saved into the local variable "opad" by "PAD_SAVE_LOCAL()"

void PAD_RESTORE_LOCAL(PAD *opad)

"PAD_SAVE_LOCAL"

Save the current pad to the local variable "opad", then make the current pad equal to "npad"

void PAD_SAVE_LOCAL(PAD *opad, PAD *npad)

"PAD_SAVE_SETNULLPAD"

Save the current pad then set it to null.

void PAD_SAVE_SETNULLPAD()

"PAD_SETSV"

Set the slot at offset "po" in the current pad to "sv"

SV * PAD_SETSV(PADOFFSET po, SV* sv)

"PAD_SET_CUR"

Set the current pad to be pad "n" in the padlist, saving the previous current pad. NB currently this macro expands to a string too long for some compilers, so it's best to replace it with

```
SAVECOMPPAD();
PAD_SET_CUR_NOSAVE(padlist,n);
```

void PAD_SET_CUR(PADLIST padlist, I32 n)

"PAD_SET_CUR_NOSAVE"

like PAD_SET_CUR, but without the save

```
void PAD_SET_CUR_NOSAVE(PADLIST padlist, I32 n)
```

"PAD_SV"

Get the value at offset "po" in the current pad

```
SV * PAD_SV(PADOFFSET po)
```

"PAD_SVI"

Lightweight and lvalue version of "PAD_SV". Get or set the value at offset "po" in the current pad. Unlike "PAD_SV", does not print diagnostics with -DX. For internal use only.

```
SV * PAD_SVI(PADOFFSET po)
```

"SAVECLEARSV"

Clear the pointed to pad value on scope exit. (i.e. the runtime action of "my")

```
void SAVECLEARSV(SV **svp)
```

"SAVECOMPPAD"

save "PL_comppad" and "PL_curpad"

```
void SAVECOMPPAD()
```

"SAVEPADSV"

Save a pad slot (used to restore after an iteration)

```
void SAVEPADSV(PADOFFSET po)
```

Password and Group access

There are only public API items currently in Password and Group access

Paths to system commands

There are only public API items currently in Paths to system commands

Prototype information

There are only public API items currently in Prototype information

REGEXP Functions

"regnode"

Described in perlreguts.

Reports and Formats

There are only public API items currently in Reports and Formats

Signals

There are only public API items currently in Signals

Site configuration

There are only public API items currently in Site configuration

Sockets configuration values

There are only public API items currently in Sockets configuration values

Source Filters

There are only public API items currently in Source Filters

Stack Manipulation Macros

"djSP"

Declare Just "SP". This is actually identical to "dSP", and declares a local copy of perl's stack pointer, available via the "SP" macro. See "'SP" in perlapi". (Available for backward source code compatibility with the old (Perl 5.005) thread model.)

```
    djSP();
```

"LVRET"

True if this op will be the return value of an lvalue subroutine

"save_alloc"

Implements "'SSNEW'" in perlapi and kin, which should be used instead of this function.

```
    I32 save_alloc(I32 size, I32 pad)
```

String Handling

"delimcpy_no_escape"

Copy a source buffer to a destination buffer, stopping at (but not including) the first occurrence in the source of the delimiter byte, "delim". The source is the bytes between "from" and "from_end" - 1. Similarly, the dest is "to" up to "to_end".

The number of bytes copied is written to *retlen.

Returns the position of "delim" in the "from" buffer, but if there is no such occurrence before

"from_end", then "from_end" is returned, and the entire buffer "from" .. "from_end" - 1 is copied.

If there is room in the destination available after the copy, an extra terminating safety "NUL" byte is appended (not included in the returned length).

The error case is if the destination buffer is not large enough to accommodate everything that should be copied. In this situation, a value larger than "to_end" - "to" is written to *retlen, and as much of the source as fits will be written to the destination. Not having room for the safety "NUL" is not considered an error.

```
char* delimcpy_no_escape(char* to, const char* to_end,
                        const char* from, const char* from_end,
                        const int delim, I32* retlen)
```

"my_cxt_init"

Implements the ""MY_CXT_INIT"" in perlxs macro, which you should use instead.

The first time a module is loaded, the global "PL_my_cxt_index" is incremented, and that value is assigned to that module's static "my_cxt_index" (whose address is passed as an arg). Then, for each interpreter this function is called for, it makes sure a "void*" slot is available to hang the static data off, by allocating or extending the interpreter's "PL_my_cxt_list" array

NOTE: "my_cxt_init" must be explicitly called as "Perl_my_cxt_init" with an "aTHX_" parameter.

```
void* Perl_my_cxt_init(pTHX_ int *indexp, size_t size)
```

"quadmath_format_needed"

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

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

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

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

"quadmath_format_valid"

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

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

Returns true if it is valid, false if not.

See also "quadmath_format_needed".

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

SV Flags

"SVt_INVLIST"

Type flag for scalars. See "svtype" in perlapi.

SV Handling

"PL_Sv"

A scratch pad SV for whatever temporary use you need. Chiefly used as a fallback by macros on platforms where "PERL_USE_GCC_BRACE_GROUPS" in perlapi> is unavailable, and which would otherwise evaluate their SV parameter more than once.

```
PL_Sv
```

"sv_2bool"

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

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

"sv_2bool_flags"

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

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

"sv_2num"

NOTE: "sv_2num" is **experimental** and may change or be removed without notice.

Return an SV with the numeric value of the source SV, doing any necessary reference or overload conversion. The caller is expected to have handled get-magic already.

```
SV* sv_2num(SV *const sv)
```

"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_add_arena"

Given a chunk of memory, link it to the head of the list of arenas, and split it into a list of free SVs.

```
void sv_add_arena(char *const ptr, const U32 size,  
                 const U32 flags)
```

"sv_clean_all"

Decrement the refcnt of each remaining SV, possibly triggering a cleanup. This function may have to be called multiple times to free SVs which are in complex self-referential hierarchies.

I32 sv_clean_all()

"sv_clean_objs"

Attempt to destroy all objects not yet freed.

void sv_clean_objs()

"sv_free_arenas"

Deallocate the memory used by all arenas. Note that all the individual SV heads and bodies within the arenas must already have been freed.

void sv_free_arenas()

"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_grow_fresh"

A cut-down version of sv_grow intended only for when sv is a freshly-minted SVt_PV, SVt_PVIV, SVt_PVNV, or SVt_PVMG. i.e. sv has the default flags, has never been any other type, and does not have an existing string. Basically, just assigns a char buffer and returns a pointer to it.

char* sv_grow_fresh(SV *const sv, STRLEN newlen)

"sv_iv"

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

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_newref"

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

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

"sv_nv"

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

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"

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

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_pvbyten_force"

The backend for the "SvPVbytex_force" macro. Always use the macro instead. If the SV cannot be downgraded from UTF-8, this croaks.

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

"sv_pvn"

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

for new code; remove it from existing code.

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_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_pvutf8"

Use the "SvPVutf8_nolen" macro instead

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

"sv_pvutf8n"

"DEPRECATED!" It is planned to remove "sv_pvutf8n" from a future release of Perl. Do not use it for new code; remove it from existing code.

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_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_tainted"

Test an SV for taintedness. Use "SvTAINTED" instead.

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

"SvTHINKFIRST"

A quick flag check to see whether an "sv" should be passed to "sv_force_normal" to be

"downgraded" before "SvIVX" or "SvPVX" can be modified directly.

For example, if your scalar is a reference and you want to modify the "SvIVX" slot, you can't just do "SvROK_off", as that will leak the referent.

This is used internally by various sv-modifying functions, such as "sv_setsv", "sv_setiv" and "sv_pvn_force".

One case that this does not handle is a gv without SvFAKE set. After

```
if (SvTHINKFIRST(gv)) sv_force_normal(gv);
```

it will still be a gv.

"SvTHINKFIRST" sometimes produces false positives. In those cases "sv_force_normal" does nothing.

```
U32 SvTHINKFIRST(SV *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_untaint"

Untaint an SV. Use "SvTAINTED_off" instead.

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

"sv_uv"

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

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)
```

Tainting

`"sv_taint"`

Taint an SV. Use `"SvTAINTED_on"` instead.

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

`"TAINT"`

If we aren't in taint checking mode, do nothing; otherwise indicate to `""TAINT_set""` and `""TAINT_PROPER""` that some unspecified element is tainted.

```
void TAIN()
```

`"TAINT_ENV"`

Looks at several components of `%ENV` for taintedness, and calls `""taint_proper""` if any are tainted. The components it searches are things like `$PATH`.

```
void TAIN_ENV
```

`"taint_env"`

Implements the `"TAINT_ENV"` macro, which you should generally use instead.

```
void taint_env()
```

`"TAINT_get"`

Returns a boolean as to whether some element is tainted or not.

```
bool TAIN_get()
```

`"TAINT_IF"`

If `"c"` evaluates to true, call `""TAINT""` to indicate that something is tainted; otherwise do nothing.

```
void TAIN_IF(bool c)
```

`"TAINTING_get"`

Returns a boolean as to whether taint checking is enabled or not.

```
bool TAINING_get()
```

`"TAINTING_set"`

Turn taint checking mode off/on

```
void TAINTING_set(bool s)
```

"TAINT_NOT"

Remove any taintedness previously set by, *e.g.*, "TAINT".

```
void TAIN_NOT()
```

"TAINT_PROPER"

If no element is tainted, do nothing; otherwise output a message (containing "s") that indicates there is a tainting violation. If such violations are fatal, it croaks.

```
void TAIN_PROPER(const char * s)
```

"taint_proper"

Implements the "TAINT_PROPER" macro, which you should generally use instead.

```
void taint_proper(const char* f, const char *const s)
```

"TAINT_set"

If "s" is true, ""TAINT_get"" returns true; If "s" is false, ""TAINT_get"" returns false;

```
void TAIN_set(bool s)
```

"TAINT_WARN_get"

Returns false if tainting violations are fatal; Returns true if they're just warnings

```
bool TAIN_WARN_get()
```

"TAINT_WARN_set"

"s" being true indicates ""TAINT_WARN_get"" should return that tainting violations are just warnings

"s" being false indicates ""TAINT_WARN_get"" should return that tainting violations are fatal.

```
void TAIN_WARN_set(bool s)
```

Time

There are only public API items currently in Time

Typedef names

There are only public API items currently in Typedef names

Unicode Support

"bytes_from_utf8_loc"

NOTE: "bytes_from_utf8_loc" is **experimental** and may change or be removed without notice.

Like ""bytes_from_utf8" in perlapi()", but takes an extra parameter, a pointer to where to store the location of the first character in "s" that cannot be converted to non-UTF8.

If that parameter is "NULL", this function behaves identically to "bytes_from_utf8".

Otherwise if *is_utf8p is 0 on input, the function behaves identically to "bytes_from_utf8", except it also sets *first_non_downgradable to "NULL".

Otherwise, the function returns a newly created "NUL"-terminated string containing the non-UTF8 equivalent of the convertible first portion of "s". *lenp is set to its length, not including the terminating "NUL". If the entire input string was converted, *is_utf8p is set to a FALSE value, and *first_non_downgradable is set to "NULL".

Otherwise, *first_non_downgradable is set to point to the first byte of the first character in the original string that wasn't converted. *is_utf8p is unchanged. Note that the new string may have length 0.

Another way to look at it is, if *first_non_downgradable is non-"NULL" and *is_utf8p is TRUE, this function starts at the beginning of "s" and converts as many characters in it as possible stopping at the first one it finds that can't be converted to non-UTF-8. *first_non_downgradable is set to point to that. The function returns the portion that could be converted in a newly created "NUL"-terminated string, and *lenp is set to its length, not including the terminating "NUL". If the very first character in the original could not be converted, *lenp will be 0, and the new string will contain just a single "NUL". If the entire input string was converted, *is_utf8p is set to FALSE and *first_non_downgradable is set to "NULL".

Upon successful return, the number of variants in the converted portion of the string can be computed by having saved the value of *lenp before the call, and subtracting the after-call value of *lenp from it.

```
U8* bytes_from_utf8_loc(const U8 *s, STRLEN *lenp,
                        bool *is_utf8p,
                        const U8 ** first_unconverted)
```

"find_uninit_var"

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

Find the name of the undefined variable (if any) that caused the operator to issue a "Use of uninitialized value" warning. If match is true, only return a name if its value matches "uninit_sv". So roughly speaking, if a unary operator (such as "OP_COS") generates a warning, then following the direct child of the op may yield an "OP_PADSV" or "OP_GV" that gives the name of the undefined variable. On the other hand, with "OP_ADD" there are two branches to follow, so we only print the variable name if we get an exact match. "desc_p" points to a string pointer holding the description of the op. This may be updated if needed.

The name is returned as a mortal SV.

Assumes that "PL_op" is the OP that originally triggered the error, and that "PL_comppad"/"PL_curpad" points to the currently executing pad.

```
SV* find_uninit_var(const OP *const obase,
                   const SV *const uninit_sv, bool match,
                   const char **desc_p)
```

"isSCRIPT_RUN"

Returns a bool as to whether or not the sequence of bytes from "s" up to but not including "send" form a "script run". "utf8_target" is TRUE iff the sequence starting at "s" is to be treated as UTF-8. To be precise, except for two degenerate cases given below, this function returns TRUE iff all code points in it come from any combination of three "scripts" given by the Unicode "Script Extensions" property: Common, Inherited, and possibly one other. Additionally all decimal digits must come from the same consecutive sequence of 10.

For example, if all the characters in the sequence are Greek, or Common, or Inherited, this function will return TRUE, provided any decimal digits in it are from the same block of digits in Common. (These are the ASCII digits "0".."9" and additionally a block for full width forms of these, and several others used in mathematical notation.) For scripts (unlike Greek) that have their own digits defined this will accept either digits from that set or from one of the Common digit sets, but not a combination of the two. Some scripts, such as Arabic, have more than one set of digits. All digits must come from the same set for this function to return TRUE.

*ret_script, if "ret_script" is not NULL, will on return of TRUE contain the script found, using the "SCX_enum" typedef. Its value will be "SCX_INVALID" if the function returns FALSE.

If the sequence is empty, TRUE is returned, but *ret_script (if asked for) will be

"SCX_INVALID".

If the sequence contains a single code point which is unassigned to a character in the version of Unicode being used, the function will return TRUE, and the script will be "SCX_Unknown". Any other combination of unassigned code points in the input sequence will result in the function treating the input as not being a script run.

The returned script will be "SCX_Inherited" iff all the code points in it are from the Inherited script.

Otherwise, the returned script will be "SCX_Common" iff all the code points in it are from the Inherited or Common scripts.

```
bool isSCRIPT_RUN(const U8 *s, const U8 *send,
                  const bool utf8_target)
```

"is_utf8_non_invariant_string"

Returns TRUE if "is_utf8_invariant_string" in perlapi returns FALSE for the first "len" bytes of the string "s", but they are, nonetheless, legal Perl-extended UTF-8; otherwise returns FALSE.

A TRUE return means that at least one code point represented by the sequence either is a wide character not representable as a single byte, or the representation differs depending on whether the sequence is encoded in UTF-8 or not.

See also ""is_utf8_invariant_string" in perlapi", ""is_utf8_string" in perlapi"

```
bool is_utf8_non_invariant_string(const U8* const s, STRLEN len)
```

"report_uninit"

Print appropriate "Use of uninitialized variable" warning.

```
void report_uninit(const SV *uninit_sv)
```

"utf8n_to_uvuni"

"DEPRECATED!" It is planned to remove "utf8n_to_uvuni" from a future release of Perl. Do not use it for new code; remove it from existing code.

Instead use "utf8_to_uvchr_buf" in perlapi, or rarely, "utf8n_to_uvchr" in perlapi.

This function was useful for code that wanted to handle both EBCDIC and ASCII platforms with

Unicode properties, but starting in Perl v5.20, the distinctions between the platforms have mostly been made invisible to most code, so this function is quite unlikely to be what you want. If you do need this precise functionality, use instead "NATIVE_TO_UNI(utf8_to_uvchr_buf(...))" or "NATIVE_TO_UNI(utf8n_to_uvchr(...))".

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

"utf8_to_uvuni"

"DEPRECATED!" It is planned to remove "utf8_to_uvuni" from a future release of Perl. Do not use it for new code; remove it from existing code.

Returns the Unicode code point of the first character in the string "s" which is assumed to be in UTF-8 encoding; "retlen" will be set to the length, in bytes, of that character.

Some, but not all, UTF-8 malformations are detected, and in fact, some malformed input could cause reading beyond the end of the input buffer, which is one reason why this function is deprecated. The other is that only in extremely limited circumstances should the Unicode versus native code point be of any interest to you. See "utf8_to_uvuni_buf" for alternatives.

If "s" points to one of the detected malformations, and UTF8 warnings are enabled, zero is returned and *retlen is set (if "retlen" doesn't point to NULL) to -1. If those warnings are off, the computed value is well-defined (or the Unicode REPLACEMENT CHARACTER, if not) is silently returned, and *retlen is set (if "retlen" isn't NULL) so that ("s" + *retlen) is the next possible position in "s" that could begin a non-malformed character. See "utf8n_to_uvchr" in perlapi for details on when the REPLACEMENT CHARACTER is returned.

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

"utf8_to_uvuni_buf"

"DEPRECATED!" It is planned to remove "utf8_to_uvuni_buf" from a future release of Perl. Do not use it for new code; remove it from existing code.

Only in very rare circumstances should code need to be dealing in Unicode (as opposed to native) code points. In those few cases, use "NATIVE_TO_UNI(utf8_to_uvchr_buf(...))" instead. If you are not absolutely sure this is one of those cases, then assume it isn't and use plain "utf8_to_uvchr_buf" instead.

Returns the Unicode (not-native) code point of the first character in the string "s" which is assumed to be in UTF-8 encoding; "send" points to 1 beyond the end of "s". "retlen" will be set to

the length, in bytes, of that character.

If "s" does not point to a well-formed UTF-8 character and UTF8 warnings are enabled, zero is returned and *retlen is set (if "retlen" isn't NULL) to -1. If those warnings are off, the computed value if well-defined (or the Unicode REPLACEMENT CHARACTER, if not) is silently returned, and *retlen is set (if "retlen" isn't NULL) so that ("s" + *retlen) is the next possible position in "s" that could begin a non-malformed character. See "utf8n_to_uvchr" in perlapi for details on when the REPLACEMENT CHARACTER is returned.

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

"uvoffuni_to_utf8_flags"

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

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

For details, see the description for "uvchr_to_utf8_flags" in perlapi.

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

"uvuni_to_utf8_flags"

"DEPRECATED!" It is planned to remove "uvuni_to_utf8_flags" from a future release of Perl. Do not use it for new code; remove it from existing code.

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

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

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

"valid_utf8_to_uvchr"

Like "'utf8_to_uvchr_buf" in perlapi", but should only be called when it is known that the next

character in the input UTF-8 string "s" is well-formed (*e.g.*, it passes ""isUTF8_CHAR" in perlapi". Surrogates, non-character code points, and non-Unicode code points are allowed.

```
UV valid_utf8_to_uvchr(const U8 *s, STRLEN *retlen)
```

"variant_under_utf8_count"

This function looks at the sequence of bytes between "s" and "e", which are assumed to be encoded in ASCII/Latin1, and returns how many of them would change should the string be translated into UTF-8. Due to the nature of UTF-8, each of these would occupy two bytes instead of the single one in the input string. Thus, this function returns the precise number of bytes the string would expand by when translated to UTF-8.

Unlike most of the other functions that have "utf8" in their name, the input to this function is NOT a UTF-8-encoded string. The function name is slightly *odd* to emphasize this.

This function is internal to Perl because khw thinks that any XS code that would want this is probably operating too close to the internals. Presenting a valid use case could change that.

See also ""is_utf8_invariant_string" in perlapi" and ""is_utf8_invariant_string_loc" in perlapi",

```
Size_t variant_under_utf8_count(const U8* const s,
                                const U8* const e)
```

Utility Functions

"my_popen_list"

Implementing function on some systems for **PerlProc_popen_list()**

```
PerlIO* my_popen_list(const char* mode, int n, SV ** args)
```

"my_socketpair"

Emulates **socketpair**(2) on systems that don't have it, but which do have enough functionality for the emulation.

```
int my_socketpair(int family, int type, int protocol, int fd[2])
```

Versioning

There are only public API items currently in Versioning

Warning and Dieing

"PL_dowarn"

The C variable that roughly corresponds to Perl's `$_^W` warning variable. However, `$_^W` is treated as a boolean, whereas "PL_dowarn" is a collection of flag bits.

On threaded perls, each thread has an independent copy of this variable; each initialized at creation time with the current value of the creating thread's copy.

U8 PL_dowarn

XS

There are only public API items currently in XS

Undocumented elements

The following functions are currently undocumented. If you use one of them, you may wish to consider creating and submitting documentation for it.

abort_execution
add_cp_to_invlist
_add_range_to_invlist
alloc_LOGOP
allocmy
amagic_cmp
amagic_cmp_desc
amagic_cmp_locale
amagic_cmp_locale_desc
amagic_is_enabled
amagic_i_ncmp
amagic_i_ncmp_desc
amagic_ncmp
amagic_ncmp_desc
any_dup
append_utf8_from_native_byte
apply
ASCII_TO_NEED
atfork_lock
atfork_unlock
av_arylen_p
av_extend_guts
av_iter_p
av_nonelem
av_reify

bind_match
block_gimme
boot_core_builtin
boot_core_mro
boot_core_PerlIO
boot_core_UNIVERSAL
_byte_dump_string
call_list
cando
cast_i32
cast_iv
cast_ulong
cast_uv
check_utf8_print
ck_anoncode
ck_backtick
ck_bitop
ck_cmp
ck_concat
ck_defined
ck_delete
ck_each
ck_entersub_args_core
ck_eof
ck_eval
ck_exec
ck_exists
ck_ftst
ck_fun
ck_glob
ck_grep
ck_index
ck_isa
ck_join
ck_length
ck_lfun
ck_listiob
ck_match
ck_method
ck_null

ck_open
ck_prototype
ck_readline
ck_reassign
ck_repeat
ck_require
ck_return
ck_rfun
ck_rvconst
ck_sassign
ck_select
ck_shift
ck_smartmatch
ck_sort
ck_spair
ck_split
ck_stringify
ck_subr
ck_substr
ck_svconst
ck_tell
ck_trunc
ck_trycatch
ckwarn
ckwarn_d
clear_defarray
closest_cop
cmpchain_extend
cmpchain_finish
cmpchain_start
cmp_desc
cmp_locale_desc
cntrl_to_mnemonic
cop_file_avn
coresub_op
create_eval_scope
croak_caller
croak_memory_wrap
croak_no_mem
croak_popstack

csighandler
csighandler1
csighandler3
current_re_engine
custom_op_get_field
cv_ckproto_len_flags
cv_clone_into
cv_const_sv_or_av
cvgv_from_hek
cvgv_set
cvstash_set
cv_undef_flags
cx_dump
cx_dup
cxinc
cx_popblock
cx_popeval
cx_popformat
cx_popgiven
cx_poploop
cx_popsb
cx_popsb_args
cx_popsb_common
cx_popwhen
cx_pushblock
cx_pusheval
cx_pushformat
cx_pushgiven
cx_pushloop_for
cx_pushloop_plain
cx_pushsb
cx_pushtry
cx_pushwhen
cx_topblock
debstackptrs
deb_stack_all
debug_hash_seed
defelem_target
delete_eval_scope
despatch_signals

die_unwind
do_aexec
do_aexec5
do_aspawn
do_eof
does_utf8_overflow
do_exec
do_exec3
dofile
do_gvkv_dump
do_gv_dump
do_hv_dump
doing_taint
do_ipcctl
do_ipcget
do_magic_dump
do_msgrcv
do_msgsnd
do_ncmp
do_open6
do_open_raw
do_op_dump
do_pmop_dump
do_print
do_readline
doref
do_seek
do_semop
do_shmio
do_spawn
do_spawn_nowait
do_sv_dump
do_sysseek
do_tell
do_trans
do_uniprop_match
do_vecget
do_vecset
do_vop
drand48_init_r

drand48_r
dtrace_probe_call
dtrace_probe_load
dtrace_probe_op
dtrace_probe_phase
dump_all_perl
dump_indent
dump_packsubs_perl
dump_sub_perl
dump_sv_child
dump_vindent
dup_warnings
emulate_cop_io
find_first_differing_byte_pos
find_lexical_cv
find_runcv_where
find_script
foldEQ_latin1
foldEQ_latin1_s2_folded
foldEQ_utf8_flags
_force_out_malformed_utf8_message
form_alien_digit_msg
form_cp_too_large_msg
free_tied_hv_pool
free_tmps
get_and_check_backslash_N_name
get_db_sub
get_debug_opts
get_deprecated_property_msg
getenv_len
get_hash_seed
get_invlist_iter_addr
get_invlist_offset_addr
get_invlist_previous_index_addr
get_mstats
get_no_modify
get_opargs
get_ppaddr
get_prop_definition
get_prop_values

get_regclass_nonbitmap_data
get_regex_charset_name
get_re_arg
get_re_gclass_nonbitmap_data
get_vtbl
gimme_V
gp_free
gp_ref
grok_bin_oct_hex
grok_bslash_c
grok_bslash_o
grok_bslash_x
gv_check
gv_fetchmeth_internal
gv_override
gv_setref
gv_stashpvn_internal
he_dup
hek_dup
hfree_next_entry
hv_auxalloc
hv_backreferences_p
hv_common
hv_common_key_len
hv_delayfree_ent
hv_kill_backrefs
hv_placeholders_p
hv_pushkv
hv_rand_set
hv_undef_flags
init_argv_symbols
init_constants
init_dbargs
init_debugger
init_i18n10n
init_i18n14n
init_named_cv
init_stacks
init_tm
init_uniprops

`_inverse_folds`
`invert`
`invlist_array`
`invlist_clear`
`invlist_clone`
`invlist_contents`
`_invlistEQ`
`invlist_extend`
`invlist_highest`
`invlist_is_iterating`
`invlist_iterfinish`
`invlist_iterinit`
`invlist_itternext`
`invlist_lowest`
`invlist_max`
`invlist_previous_index`
`invlist_set_len`
`invlist_set_previous_index`
`invlist_trim`
`_invlist_array_init`
`_invlist_contains_cp`
`_invlist_dump`
`_invlist_intersection`
`_invlist_intersection_maybe_complement_2nd`
`_invlist_invert`
`_invlist_len`
`_invlist_search`
`_invlist_subtract`
`_invlist_union`
`_invlist_union_maybe_complement_2nd`
`invmap_dump`
`io_close`
`isFF_overlong`
`is_grapheme`
`is_invlist`
`is_utf8_char_helper_`
`is_utf8_common`
`is_utf8_FF_helper_`
`is_utf8_overlong`
`_is_cur_LC_category_utf8`

`_is_in_locale_category`
`_is_uni_FOO`
`_is_uni_perl_idcont`
`_is_uni_perl_idstart`
`_is_utf8_FOO`
`_is_utf8_perl_idcont`
`_is_utf8_perl_idstart`
`jmaybe`
`keyword`
`keyword_plugin_standard`
`list`
`load_charnames`
`localize`
`lossless_NV_to_IV`
`lsbit_pos32`
`lsbit_pos64`
`magic_cleararylen_p`
`magic_clearenv`
`magic_clearisa`
`magic_clearpack`
`magic_clearsig`
`magic_clear_all_env`
`magic_copycallchecker`
`magic_existspack`
`magic_freearylen_p`
`magic_freecollxfrm`
`magic_freemglob`
`magic_freeovrld`
`magic_freeutf8`
`magic_get`
`magic_getarylen`
`magic_getdebugvar`
`magic_getdefelem`
`magic_getnkeys`
`magic_getpack`
`magic_getpos`
`magic_getsig`
`magic_getsubstr`
`magic_gettaint`
`magic_getuvar`

magic_getvec
magic_killbackrefs
magic_nextpack
magic_regdata_cnt
magic_regdatum_get
magic_regdatum_set
magic_scalarpack
magic_set
magic_setarylen
magic_setcollxfrm
magic_setdblne
magic_setdebugvar
magic_setdefelem
magic_setenv
magic_setisa
magic_setlref
magic_setmglob
magic_setnkeys
magic_setnonelem
magic_setpack
magic_setpos
magic_setregexp
magic_setsig
magic_setsigall
magic_setsubstr
magic_settaint
magic_setutf8
magic_setuvar
magic_setvec
magic_set_all_env
magic_sizepack
magic_wipepack
malloced_size
malloc_good_size
markstack_grow
mem_collxfrm
mem_log_alloc
mem_log_del_sv
mem_log_free
mem_log_new_sv

mem_log_realloc
_mem_collxfrm
mg_find_mglob
mg_size
mode_from_discipline
more_bodies
more_sv
moreswitches
mortal_getenv
mro_get_private_data
mro_meta_dup
mro_meta_init
msbit_pos32
msbit_pos64
multiconcat_stringify
multideref_stringify
my_atof2
my_atof3
my_attrs
my_clearenv
my_lstat
my_lstat_flags
my_memchr
my_mkostemp_cloexec
my_mkstemp_cloexec
my_stat
my_stat_flags
my_strerror
my_unexec
NATIVE_TO_NEED
newFORM
newGP
newMETHOP_internal
newMYSUB
newPROG
new_stackinfo
newSTUB
newSVavdefelem
new_warnings_bitfield
newXS_deffile

_new_invlist
_new_invlist_C_array
nextargv
no_bareword_filehandle
noperl_die
notify_parser_that_changed_to_utf8
oopsAV
oopsHV
op_clear
op_integerize
op_lvalue_flags
opmethod_stash
op_refcnt_dec
op_refcnt_inc
op_relocate_sv
opslab_force_free
opslab_free
opslab_free_nopad
op_std_init
op_unscope
package
package_version
pad_add_weakref
padlist_store
padname_free
PadnameIN_SCOPE
padnamelist_free
parser_dup
parser_free
parser_free_nexttoke_ops
parse_unicode_opts
path_is_searchable
peep
perl_alloc_using
perl_clone_using
PerlIO_context_layers
PerlIO_restore_errno
PerlIO_save_errno
PerlLIO_dup2_cloexec
PerlLIO_dup_cloexec

PerlLIO_open3_cloexec
PerlLIO_open_cloexec
PerlProc_pipe_cloexec
PerlSock_accept_cloexec
PerlSock_socketpair_cloexec
PerlSock_socket_cloexec
perly_sighandler
pmsruntime
POPMARK
populate_isa
pregfree
pregfree2
qerror
ReANY
reentrant_free
reentrant_init
reentrant_retry
reentrant_size
re_exec_indentf
ref
regcurly
regdump
regdupe_internal
regexec_flags
regfree_internal
reginitcolors
reg_named_buff
reg_named_buff_all
reg_named_buff_exists
reg_named_buff_fetch
reg_named_buff_firstkey
reg_named_buff_iter
reg_named_buff_nextkey
reg_named_buff_scalar
regnext
reg_numbered_buff_fetch
reg_numbered_buff_length
reg_numbered_buff_store
regprop
reg_qr_package

reg_skipcomment
reg_temp_copy
re_indentf
re_intuit_start
re_intuit_string
re_op_compile
report_evil_fh
report_redefined_cv
report_wrongway_fh
re_printf
rpeek
rsignal_restore
rsignal_save
rvpv_dup
rxres_save
same_dirent
save_bool
save_clearsv
save_delete
save_destructor
save_destructor_x
save_freeop
save_freepv
save_freesv
save_I16
save_I32
save_I8
save_int
save_iv
save_long
save_mortalizesv
save_pptr
save_re_context
save_sptr
savestack_grow
savestack_grow_cnt
save_strlen
save_to_buffer
sawparens
scalar

scalarvoid
scan_num
scan_str
scan_word
seed
set_caret_X
setfd_cloexec
setfd_cloexec_for_nonsysfd
setfd_cloexec_or_inhexec_by_sysfdness
setfd_inhexec
setfd_inhexec_for_sysfd
set_numeric_standard
set_numeric_underlying
set_padlist
_setup_canned_invlst
share_hek
should_warn_nl
should_we_output_Debug_r
sighandler
sighandler1
sighandler3
single_1bit_pos32
single_1bit_pos64
skipspace_flags
Slab_Alloc
Slab_Free
Slab_to_ro
Slab_to_rw
softref2xv
sortsv_flags_impl
stack_grow
str_to_version
sub_crush_depth
sv_2iv
sv_2uv
sv_add_backref
sv_buf_to_ro
sv_del_backref
sv_free2
sv_i_ncmp

sv_i_ncmp_desc
sv_kill_backrefs
sv_magicext_mglob
sv_ncmp
sv_ncmp_desc
sv_only_taint_gmagic
sv_or_pv_pos_u2b
sv_resetpvn
sv_sethek
sv_setsv_cow
SvTRUE_common
sv_unglob
sys_init
sys_init3
sys_intern_clear
sys_intern_dup
sys_intern_init
sys_term
tied_method
tmpls_grow_p
TOPMARK
to_uni_fold
to_uni_lower
to_uni_title
to_uni_upper
_to_fold_latin1
_to_uni_fold_flags
_to_upper_title_latin1
_to_utf8_fold_flags
_to_utf8_lower_flags
_to_utf8_title_flags
_to_utf8_upper_flags
translate_substr_offsets
try_amagic_bin
try_amagic_un
uiv_2buf
unlnk
unshare_hek
utf16_to_utf8
utf16_to_utf8_base

utf16_to_utf8_reversed
_utf8n_to_uvchr_msgs_helper
utf8_to_utf16_base
utf8_to_uvchr_buf_helper
utilize
uvoffuni_to_utf8_flags_msgs
uvuni_to_utf8
valid_utf8_to_uvuni
variant_byte_number
varname
vivify_defelem
vivify_ref
wait4pid
_warn_problematic_locale
was_lvalue_sub
watch
win32_croak_not_implemented
write_to_stderr
xs_boot_epilog
xs_handshake
yyerror
yyerror_pv
yyerror_pvn
yylex
yyparse
yyquit
yyunlex

AUTHORS

The autodocumentation system was originally added to the Perl core by Benjamin Stuhl. Documentation is by whoever was kind enough to document their functions.

SEE ALSO

config.h, perlapi, perlapi, perlcalls, perlclib, perlembed, perlfilters, perlguts, perlhacktips, perlinterp, perliol, perlroapi, perlreapi, perlreguts, perlxs