

## 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!**

## Compile-time scope hooks

### BhkENTRY

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

Return the BHK's flags.

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

### CALL\_BLOCK\_HOOKS

NOTE: this function 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)
```

## 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 Manipulation Functions

### 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)
```

## CV reference counts and CvOUTSIDE

### 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)
```

## Embedding Functions

### `cv_dump`

dump the contents of a CV

```
void cv_dump(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)
```

#### 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.

```
void 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\_swipe

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

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

## GV Functions

#### gv\_try\_downgrade

NOTE: this function 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)
```

## Hash Manipulation Functions

### `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)
```

### `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_pvn`

Like *refcounted\_he\_fetch\_pvn*, but takes a nul-terminated string instead of a string/length pair.

```
SV * refcounted_he_fetch_pvn(
    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 NUL-terminated literal string instead of a string/length pair, and no precomputed hash.

```
SV * refcounted_he_fetch_pvs(
    const struct refcounted_he *chain,
    const char *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\_pv

Like *refcounted\_he\_new\_pvn*, but takes a nul-terminated string instead of a string/length pair.

```
struct refcounted_he * refcounted_he_new_pv(
    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 NUL-terminated literal string instead of a string/length pair, and no precomputed hash.

```
struct refcounted_he * refcounted_he_new_pvs(
    struct refcounted_he *parent,
    const char *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
)
```

## IO Functions

`start_glob`

NOTE: this function 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.

```
PerlIO* start_glob(SV *tmpglob, IO *io)
```

## Lexer interface

`validate_proto`

NOTE: this function 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`.

NOTE: the `perl_` form of this function is deprecated.

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

## Magical Functions

### `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:

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

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

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

```
SV* magic_methcall(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_localize`

Copy some of the magic from an existing SV to new localized version of that SV.

Container magic (e.g., `%ENV`, `$!`, `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)
```



## Miscellaneous Functions

### `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)
```

## MRO Functions

### `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)
```

## Optree Manipulation Functions

### `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)
```

## 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()
```

#### 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\_SVl

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_SVl(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)

XXX DAPM it would make more sense to make the arg a PADOFFSET void

```
SAVEPADSV(PADOFFSET po)
```

## Per-Interpreter Variables

#### 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*.

```
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*.

```
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*.

```
SV * PL_DBtrace
```

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.

```
U8 PL_dowarn
```

PL\_last\_in\_gv

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

```
GV* PL_last_in_gv
```

PL\_ofsgv

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

```
GV* PL_ofsgv
```

PL\_rs

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

```
SV* PL_rs
```

## 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

## SV-Body Allocation

sv\_2num

NOTE: this function 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 Manipulation Functions

An SV (or AV, HV, etc.) is allocated in two parts: the head (struct `sv`, `av`, `hv`...) contains type and reference count information, and for many types, a pointer to the body (struct `xrv`, `xpv`, `xpviv`...), which contains fields specific to each type. Some types store all they need in the head, so don't have a body.

In all but the most memory-paranoid configurations (ex: `PURIFY`), heads and bodies are allocated out of arenas, which by default are approximately 4K chunks of memory parcelled up into N heads or bodies. Sv-bodies are allocated by their sv-type, guaranteeing size consistency needed to allocate safely from arrays.

For SV-heads, the first slot in each arena is reserved, and holds a link to the next arena, some flags, and a note of the number of slots. Snaked through each arena chain is a linked list of free items; when this becomes empty, an extra arena is allocated and divided up into N items which are threaded into the free list.

SV-bodies are similar, but they use arena-sets by default, which separate the link and info from the arena itself, and reclaim the 1st slot in the arena. SV-bodies are further described later.

The following global variables are associated with arenas:

<code>PL_sv_arenaroot</code>	pointer to list of SV arenas
<code>PL_sv_root</code>	pointer to list of free SV structures
<code>PL_body_arenas</code>	head of linked-list of body arenas
<code>PL_body_roots[]</code>	array of pointers to list of free bodies of svtype arrays are indexed by the svtype needed

A few special SV heads are not allocated from an arena, but are instead directly created in the interpreter structure, eg `PL_sv_undef`. The size of arenas can be changed from the default by setting `PERL_ARENA_SIZE` appropriately at compile time.

The SV arena serves the secondary purpose of allowing still-live SVs to be located and destroyed during final cleanup.

At the lowest level, the macros `new_SV()` and `del_SV()` grab and free an SV head. (If debugging with `-DD`, `del_SV()` calls the function `S_del_sv()` to return the SV to the free list with error checking.) `new_SV()` calls `more_sv()` / `sv_add_arena()` to add an extra arena if the free list is empty. SVs in the free list have their `SvTYPE` field set to all ones.

At the time of very final cleanup, `sv_free_arenas()` is called from `perl_destruct()` to physically free all the arenas allocated since the start of the interpreter.

The function `visit()` scans the SV arenas list, and calls a specified function for each SV it finds which is still live - ie which has an `SvTYPE` other than all 1's, and a non-zero `SvREFCNT`. `visit()` is used by the following functions (specified as [function that calls `visit()`] / [function called by `visit()` for each SV]):

```
sv_report_used() / do_report_used()
dump all remaining SVs (debugging aid)

sv_clean_objs() / do_clean_objs(), do_clean_named_objs(),
do_clean_named_io_objs(), do_curse()
Attempt to free all objects pointed to by RVs,
try to do the same for all objects indir-
ectly referenced by typeglobs too, and
then do a final sweep, cursing any
objects that remain. Called once from
perl_destruct(), prior to calling sv_clean_all()
below.

sv_clean_all() / do_clean_all()
SvREFCNT_dec(sv) each remaining SV, possibly
triggering an sv_free(). It also sets the
SVf_BREAK flag on the SV to indicate that the
refcnt has been artificially lowered, and thus
stopping sv_free() from giving spurious warnings
about SVs which unexpectedly have a refcnt
of zero. called repeatedly from perl_destruct()
```

until there are no SVs left.

#### 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()
```

#### 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)
```

## Unicode Support

#### find\_uninit\_var

NOTE: this function 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)
```

`report_uninit`

Print appropriate "Use of uninitialized variable" warning.

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

## Undocumented functions

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

```
PerlIO_restore_errno
PerlIO_save_errno
Slab_Alloc
Slab_Free
Slab_to_ro
Slab_to_rw
_add_range_to_invlist
_byte_dump_string
_core_swash_init
_get_regclass_nonbitmap_data
_get_swash_invlist
_invlistEQ
_invlist_array_init
_invlist_contains_cp
_invlist_dump
_invlist_intersection
_invlist_intersection_maybe_complement_2nd
_invlist_invert
_invlist_len
_invlist_populate_swash
_invlist_search
_invlist_subtract
_invlist_union
_invlist_union_maybe_complement_2nd
_is_grapheme
_load_PL_utf8_foldclosures
_mem_collxfrm
_new_invlist
```



`_new_invlst_C_array`  
`_setup_canned_invlst`  
`_swash_inversion_hash`  
`_swash_to_invlst`  
`_to_fold_latin1`  
`_to_upper_title_latin1`  
`_warn_problematic_locale`  
`abort_execution`  
`add_cp_to_invlst`  
`alloc_LOGOP`  
`alloc_maybe_populate_EXACT`  
`allocmy`  
`amagic_is_enabled`  
`append_utf8_from_native_byte`  
`apply`  
`av_extend_guts`  
`av_reify`  
`bind_match`  
`boot_core_PerlIO`  
`boot_core_UNIVERSAL`  
`boot_core_mro`  
`cando`  
`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_join`

ck\_length  
ck\_ifun  
ck\_listiob  
ck\_match  
ck\_method  
ck\_null  
ck\_open  
ck\_prototype  
ck\_readline  
ck\_refassign  
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  
closest\_cop  
compute\_EXACTish  
coresub\_op  
create\_eval\_scope  
croak\_caller  
croak\_no\_mem  
croak\_popstack  
current\_re\_engine  
custom\_op\_get\_field  
cv\_ckproto\_len\_flags  
cv\_clone\_into  
cv\_const\_sv\_or\_av  
cv\_undef\_flags  
cvgv\_from\_hek

cvgv\_set  
cvstash\_set  
deb\_stack\_all  
defelem\_target  
delete\_eval\_scope  
delimcpy\_no\_escape  
die\_unwind  
do\_aexec  
do\_aexec5  
do\_eof  
do\_exec  
do\_exec3  
do\_execfree  
do\_ipcctl  
do\_ipcget  
do\_msgrcv  
do\_msgsnd  
do\_ncmp  
do\_open6  
do\_open\_raw  
do\_print  
do\_readline  
do\_seek  
do\_semop  
do\_shmio  
do\_sysseek  
do\_tell  
do\_trans  
do\_vecget  
do\_vecset  
do\_vop  
does\_utf8\_overflow  
dofile  
drand48\_init\_r  
drand48\_r  
dtrace\_probe\_call  
dtrace\_probe\_load  
dtrace\_probe\_op  
dtrace\_probe\_phase  
dump\_all\_perl  
dump\_packsubs\_perl  
dump\_sub\_perl

dump\_sv\_child  
emulate\_cop\_io  
feature\_is\_enabled  
find\_lexical\_cv  
find\_runcv\_where  
find\_script  
form\_short\_octal\_warning  
free\_tied\_hv\_pool  
get\_db\_sub  
get\_debug\_opts  
get\_hash\_seed  
get\_invlist\_iter\_addr  
get\_invlist\_offset\_addr  
get\_invlist\_previous\_index\_addr  
get\_no\_modify  
get\_opargs  
get\_re\_arg  
getenv\_len  
grok\_atoUV  
grok\_bslash\_c  
grok\_bslash\_o  
grok\_bslash\_x  
gv\_fetchmeth\_internal  
gv\_override  
gv\_setref  
gv\_stashpv\_n\_internal  
gv\_stashsvpv\_n\_cached  
handle\_named\_backref  
hfree\_next\_entry  
hv\_backreferences\_p  
hv\_kill\_backrefs  
hv\_placeholders\_p  
hv\_undef\_flags  
init\_argv\_symbols  
init\_constants  
init\_dbargs  
init\_debugger  
invert  
invlist\_array  
invlist\_clear  
invlist\_clone  
invlist\_highest

invlist\_is\_iterating  
invlist\_iterfinish  
invlist\_iterinit  
invlist\_max  
invlist\_previous\_index  
invlist\_set\_len  
invlist\_set\_previous\_index  
invlist\_trim  
io\_close  
isFF\_OVERLONG  
isFOO\_lc  
is\_utf8\_common  
is\_utf8\_common\_with\_len  
is\_utf8\_cp\_above\_31\_bits  
is\_utf8\_overlong\_given\_start\_byte\_ok  
isinfnansv  
jmaybe  
keyword  
keyword\_plugin\_standard  
list  
localize  
magic\_clear\_all\_env  
magic\_cleararylen\_p  
magic\_clearenv  
magic\_clearisa  
magic\_clearpack  
magic\_clearsig  
magic\_copycallchecker  
magic\_existspack  
magic\_freearylen\_p  
magic\_freeovrld  
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\_set\_all\_env  
magic\_setarylen  
magic\_setcollxfrm  
magic\_setdbline  
magic\_setdebugvar  
magic\_setdefelem  
magic\_setenv  
magic\_setisa  
magic\_setlvref  
magic\_setmglob  
magic\_setnkeys  
magic\_setpack  
magic\_setpos  
magic\_setregexp  
magic\_setsig  
magic\_setsubstr  
magic\_settaint  
magic\_setutf8  
magic\_setuvar  
magic\_setvec  
magic\_sizepack  
magic\_wipepack  
malloc\_good\_size  
malloced\_size  
mem\_collxfrm  
mem\_log\_alloc  
mem\_log\_free  
mem\_log\_realloc  
mg\_find\_mglob  
mode\_from\_discipline  
more\_bodies  
mro\_meta\_dup  
mro\_meta\_init  
multideref\_stringify

my\_attrs  
my\_clearenv  
my\_lstat\_flags  
my\_stat\_flags  
my\_unexec  
newATTRSUB\_x  
newGP  
newMETHOP\_internal  
newSTUB  
newSVavdefelem  
newXS\_deffile  
newXS\_len\_flags  
new\_warnings\_bitfield  
nextargv  
noperl\_die  
notify\_parser\_that\_changed\_to\_utf8  
oopsAV  
oopsHV  
op\_clear  
op\_integerize  
op\_lvalue\_flags  
op\_refcnt\_dec  
op\_refcnt\_inc  
op\_relocate\_sv  
op\_std\_init  
op\_unscope  
opmethod\_stash  
opslab\_force\_free  
opslab\_free  
opslab\_free\_nopad  
package  
package\_version  
pad\_add\_weakref  
padlist\_store  
padname\_free  
padnamelist\_free  
parse\_unicode\_opts  
parser\_free  
parser\_free\_nexttoke\_ops  
path\_is\_searchable  
peep  
pmruntime

populate\_isa  
ptr\_hash  
qerror  
re\_exec\_indentf  
re\_indentf  
re\_op\_compile  
re\_printf  
reg\_named\_buff  
reg\_named\_buff\_iter  
reg\_numbered\_buff\_fetch  
reg\_numbered\_buff\_length  
reg\_numbered\_buff\_store  
reg\_qr\_package  
reg\_skipcomment  
reg\_temp\_copy  
regcurly  
regprop  
report\_evil\_fh  
report\_redefined\_cv  
report\_wrongway\_fh  
rpeek  
rsignal\_restore  
rsignal\_save  
rxres\_save  
same\_dirent  
save\_strlen  
sawparens  
scalar  
scalarvoid  
set\_caret\_X  
set\_padlist  
should\_warn\_nl  
sighandler  
softref2xv  
ssc\_add\_range  
ssc\_clear\_locale  
ssc\_cp\_and  
ssc\_intersection  
ssc\_union  
sub\_crush\_depth  
sv\_add\_backref  
sv\_buf\_to\_ro



sv\_del\_backref  
sv\_free2  
sv\_kill\_backrefs  
sv\_len\_utf8\_nomg  
sv\_magicext\_mglob  
sv\_mortalcopy\_flags  
sv\_only\_taint\_gmagic  
sv\_or\_pv\_pos\_u2b  
sv\_resetpvn  
sv\_sethek  
sv\_setsv\_cow  
sv\_unglob  
swash\_fetch  
swash\_init  
tied\_method  
tmps\_grow\_p  
translate\_substr\_offsets  
try\_amagic\_bin  
try\_amagic\_un  
unshare\_hek  
utilize  
varname  
vivify\_defelem  
vivify\_ref  
wait4pid  
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**

*perlguts*, *perlapi*