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)
```

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 a 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, eg

```
    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)
```

intro_my

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

```
 U32 intro_my()
```

padlist_dup

Duplicates a pad.

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

pad_alloc_name

Allocates a place in the currently-compiling pad (via *"pad_alloc" in perlapi*) and then stores a name for that entry. *namesv* is adopted and becomes the name entry; it must already contain the name string and be sufficiently upgraded. *typestash* and *ourstash* and the `padadd_STATE` flag get added to *namesv*. 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(SV *namesv, 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 C<ourstash>
```

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

```
void pad_check_dup(SV *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_sv is set to the innermost matched namesv or fake namesv; out_flags returns the flags normally associated with the IVX field of a fake namesv.

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 namesvs in anon protoypes have to store in xlow 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,
                      SV** out_name_sv, 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()
```

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)
```

Functions in file op.c

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)
```

Functions in file pp_ctl.c

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(OP *o)
```

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_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,
     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 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. 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:

```
    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.

```
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 (eg %ENV, $1, tie) gets copied, value magic doesn't (eg 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)
```

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)
```

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.

```
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_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). Note that `SvUVX` is hijacked for this purpose.

```
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`. Note that `SvUV_set` is hijacked for this purpose.

```
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)
```

pad_peg

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

When PERL_MAD is enabled, this is a small no-op function that gets called at the start of each pad-related function. It can be breakpointed to track all pad operations. The parameter is a string indicating the type of pad operation being performed.

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

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

```
void 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 which corresponds to Perl's $^W warning variable.

```
bool 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. (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 Manipulation Functions

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_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-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. You must use the `SvNUM(sv)` macro to access this function.

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

sv_copypv

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

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

sv_ref

Returns a SV describing what the SV passed in is a reference to.

```
SV* sv_ref(SV *dst, const SV *const sv,
        const int ob)
```

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.

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 top)
```

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.

Slab_Alloc

Slab_Free

Slab_to_ro

Slab_to_rw

_add_range_to_invlist

_core_swash_init

_get_swash_invlist

_invlist_array_init

_invlist_contains_cp

_invlist_contents

_invlist_dump

_invlist_intersection

_invlist_intersection_maybe_complement_2nd

_invlist_invert

_invlist_len

_invlist_populate_swatch

_invlist_search

_invlist_subtract

_invlist_union

_invlist_union_maybe_complement_2nd

_new_invlist

_setup_canned_invlist

_swash_inversion_hash

_swash_to_invlist

_to_fold_latin1

_to_upper_title_latin1

aassign_common_vars

add_cp_to_invlist

addmad

alloc_maybe_populate_EXACT

allocmy

amagic_is_enabled

append_madprops

apply

av_extend_guts

av_reify

bind_match

block_end

block_start

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_lfun

ck_listiob

ck_match

ck_method

ck_null

ck_open

ck_readline

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_subr

ck_substr

ck_svconst

ck_tell

ck_trunc

closest_cop

compute_EXACTish

convert

coresub_op

create_eval_scope

croak_no_mem

croak_popstack

current_re_engine

custom_op_get_field

cv_ckproto_len_flags

cv_clone_into

cv_const_sv_or_av

cvgv_set

cvstash_set

deb_stack_all

defelem_target

delete_eval_scope

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_op_xmldump

do_open6

do_open_raw

do_pmop_xmldump

do_print

do_readline

do_seek

do_semop

do_shmio

do_sysseek

do_tell

do_trans

do_vecget

do_vecset

do_vop

dofile

drand48_init_r

drand48_r

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_rundefsv2

find_script

free_tied_hv_pool

get_and_check_backslash_N_name

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_bslash_x

gv_override

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_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

isALNUM_lazy

isIDFIRST_lazy

is_utf8_char_slow

is_utf8_common

jmaybe

keyword

keyword_plugin_standard

list

localize

mad_free

madlex

madparse

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_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_setdefelem
magic_setenv
magic_setisa
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
mg_find_mglob
mode_from_discipline

more_bodies
mro_meta_dup
mro_meta_init
my_attrs
my_clearenv
my_lstat_flags
my_stat_flags
my_unexec
newATTRSUB_x
newGP
newMADPROP
newMADsv
newSTUB
newSVavdefelem
newTOKEN
newXS_len_flags
new_warnings_bitfield
nextargv
oopsAV
oopsHV
op_clear
op_const_sv
op_getmad
op_getmad_weak
op_integerize
op_lvalue_flags
op_refcnt_dec
op_refcnt_inc
op_std_init
op_unscope
op_xmldump
opslab_force_free
opslab_free
opslab_free_nopad
package
package_version
padlist_store
parse_subsignature
parse_unicode_opts
parser_free
parser_free_nexttoke_ops
path_is_searchable

peep

pmop_xmldump

pmruntime

populate_isa

prepend_madprops

ptr_hash

qerror

re_op_compile

reg_named_buff

reg_named_buff_iter

reg_numbered_buff_fetch

reg_numbered_buff_length

reg_numbered_buff_store

reg_qr_package

reg_temp_copy

regcurly

regpposixcc

regprop

reguni

report_evil_fh

report_redefined_cv

report_wrongway_fh

rpeep

rsignal_restore

rsignal_save

rxres_save

same_dirent

save_strlen

sawparens

scalar

scalarvoid

set_caret_X

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_catxmlpv

sv_catxmlpvn

sv_catxmlsv

sv_del_backref

sv_free2

sv_kill_backrefs

sv_len_utf8_nomg

sv_magicext_mglob

sv_mortalcopy_flags

sv_or_pv_pos_u2b

sv_resetpvn

sv_sethek

sv_setsv_cow

sv_unglob

sv_xmlpeek

tied_method

token_free

token_getmad

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

xmldump_all

xmldump_all_perl

xmldump_eval

xmldump_form

xmldump_indent

xmldump_packsubs

xmldump_packsubs_perl

xmldump_sub

xmldump_sub_perl

xmldump_vindent

xs_apiversion_bootcheck

xs_version_bootcheck

yyerror

yyerror_pv

yyerror_pvn

yylex

yyparse

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*