

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!**

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 (ie 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)
```

Functions in file pad.h**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_PARAMS* param` Clone the state variables associated with running and compiling pads.

```
void PAD_CLONE_VARS(PerlInterpreter *proto_perl \)
```

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 `svCUR` is hijacked for this purpose.

```
STRLEN PAD_COMPNAME_GEN(PADOFFSET po)
```

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_DUP

Clone a padlist.

```
void PAD_DUP(PADLIST dstpad, PADLIST srcpad, CLONE_PARAMS* param)
```

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.

```
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_SV1

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_SV1 (PADOFFSET po)
```

SAVECLEARSV

Clear the pointed to pad value on scope exit. (ie 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)`

Functions in file pp_ctl.c

find_runcv

Locate the CV corresponding to the currently executing sub or eval. If db_seqp is non_null, skip CVs that are in the DB package and populate *db_seqp with the cop sequence number at the point that the DB:: code was entered. (allows debuggers to eval in the scope of the breakpoint rather than in in the scope of the debugger itself).

```
CV* find_runcv(U32 *db_seqp)
```

Global 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_ofs_sv

The output field separator - \$, in Perl space.

```
SV* PL_ofs_sv
```

PL_rs

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

```
SV* PL_rs
```

GV Functions

is_gv_magical

Returns TRUE if given the name of a magical GV.

Currently only useful internally when determining if a GV should be created even in rvalue contexts.

`flags` is not used at present but available for future extension to allow selecting particular classes of magical variable.

Currently assumes that `name` is NUL terminated (as well as `len` being valid). This assumption is met by all callers within the perl core, which all pass pointers returned by `SvPV`.

```
bool is_gv_magical(char *name, STRLEN len, U32 flags)
```

IO Functions

start_glob

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* pattern, IO *io)
```

Pad Data Structures

CvPADLIST

CV's can have `CvPADLIST(cv)` set to point to an AV.

For these purposes "forms" are a kind-of CV, `eval`'s are too (except they're not callable at will and are always thrown away after the `eval` is done executing).

XSUBs don't have `CvPADLIST` set - `dXSTARG` fetches values from `PL_curpad`, but that is really the callers pad (a slot of which is allocated by every `entersub`).

The `CvPADLIST` AV has does not have `AvREAL` set, so `REFCNT` of component items is managed "manual" (mostly in `pad.c`) rather than normal `av.c` rules. The items in the AV are not SVs as for a normal AV, but other AVs:

0'th Entry of the `CvPADLIST` is an AV which represents the "names" or rather the "static type information" for lexicals.

The `CvDEPTH`'th entry of `CvPADLIST` AV is an AV which is the stack frame at that depth of recursion into the CV. The 0'th slot of a frame AV is an AV which is `@_`. other entries are storage for variables and op targets.

During compilation: `PL_comppad_name` is set to the names AV. `PL_comppad` is set to the frame AV for the frame `CvDEPTH == 1`. `PL_curpad` is set to the body of the frame AV (i.e. `AvARRAY(PL_comppad)`).

During execution, `PL_comppad` and `PL_curpad` refer to the live frame of the currently executing sub.

Iterating over the names AV iterates over all possible pad items. Pad slots that are `SVs_PADTMP` (targets/GVs/constants) end up having `&PL_sv_undef` "names" (see `pad_alloc()`).

Only `my/our` variable (`SVs_PADMY/SVs_PADOUR`) slots get valid names. The rest are op targets/GVs/constants which are statically allocated or resolved at compile time. These don't have names by which they can be looked up from Perl code at run time through `eval` like `my/our` variables can be. Since they can't be looked up by "name" but only by their index allocated at compile time (which is usually in `PL_op->op_targ`), wasting a name SV for them doesn't make sense.

The SVs in the names AV have their PV being the name of the variable. `NV+1..IV` inclusive is a range of `cop_seq` numbers for which the name is valid. For typed lexicals name SV is `SVt_PVMG` and `SvSTASH` points at the type. For `our` lexicals, the type is

SVt_PVGV, and GvSTASH points at the stash of the associated global (so that duplicate `our` declarations in the same package can be detected). SvCUR is sometimes hijacked to store the generation number during compilation.

If SvFAKE is set on the name SV then slot in the frame AVs are a REFCNT'ed references to a lexical from "outside". In this case, the name SV does not have a `cop_seq` range, since it is in scope throughout.

If the 'name' is '&' the corresponding entry in frame AV is a CV representing a possible closure. (SvFAKE and name of '&' is not a meaningful combination currently but could become so if `my sub foo {}` is implemented.)

The flag SVf_PADSTALE is cleared on lexicals each time the `my()` is executed, and set on scope exit. This allows the 'Variable \$x is not available' warning to be generated in evals, such as

```
{ my $x = 1; sub f { eval '$x' } } f();
```

```
AV * CvPADLIST(CV *cv)
```

cv_clone

Clone a CV: make a new CV which points to the same code etc, but which has a newly-created pad built by copying the prototype pad and capturing any outer lexicals.

```
CV* cv_clone(CV* proto)
```

cv_dump

dump the contents of a CV

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

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.

```
U32 intro_my()
```

pad_add_anon

Add an anon code entry to the current compiling pad

```
PADOFFSET pad_add_anon(SV* sv, OPCODE op_type)
```

pad_add_name

Create a new name in the current pad at the specified offset. If `typestash` is valid, the name is for a typed lexical; set the name's stash to that value. If `ourstash` is valid, it's an `our` lexical, set the name's GvSTASH to that value

Also, if the name is `@..` or `%..`, create a new array or hash for that slot

If fake, it means we're cloning an existing entry

```
PADOFFSET pad_add_name(char *name, HV* typestash, HV* ourstash, bool clone)
```

pad_alloc

Allocate a new my or tmp pad entry. For a my, simply push a null SV onto the end of PL_comppad, but for a tmp, scan the pad from PL_padix upwards for a slot which has no name and no active value.

```
PADOFFSET pad_alloc(I32 optype, U32 tmptype)
```

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(char* name, bool is_our, 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. innercv is the CV *inside* the chain of outer CVs to be searched. If newoff is non-null, this is a run-time cloning: don't add fake entries, just find the lexical and add a ref to it at newoff in the current pad.

```
PADOFFSET pad_findlex(char* name, PADOFFSET newoff, CV* innercv)
```

pad_findmy

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

```
PADOFFSET pad_findmy(char* name)
```

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_new

Create a new compiling padlist, saving and updating the various global vars at the

same time as creating the pad itself. The following flags can be OR'ed together:

```
padnew_CLONE  this pad is for a cloned CV
padnew_SAVE   save old globals
padnew_SAVESUB also save extra stuff for start of sub
```

```
PADLIST* pad_new(int flags)
```

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. If `has_args` is true, give the new pad an `@_` in slot zero.

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

pad_reset

Mark all the current temporaries for reuse

```
void pad_reset()
```

pad_setsv

Set the entry at offset `po` in the current pad to `sv`. Use the macro `PAD_SETSV()` rather than calling this function directly.

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

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)
```

pad_tidy

Tidy up a pad after we've finished compiling it: * remove most stuff from the pads of anonsub prototypes; * give it a `@_`; * mark tmps as such.

```
void pad_tidy(padtidy_type type)
```

pad_undef

Free the padlist associated with a CV. If parts of it happen to be current, we null the relevant `PL_*pad*` global vars so that we don't have any dangling references left. We also repoint the `CvOUTSIDE` of any about-to-be-orphaned inner subs to the outer of this cv.

(This function should really be called `pad_free`, but the name was already taken)

```
void pad_undef(CV* cv)
```

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

report_uninit

Print appropriate "Use of uninitialized variable" warning

```
void report_uninit()
```

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* ptr, U32 size, 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()
```

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(1), perlapi(1)