# PerlGuts Illustrated
Version 0.09

This document is meant to supplement the *perlguts(1)* manual page that comes with Perl. It contains commented illustrations of all (eventually?) major internal Perl data structures. Having this document handy hopefully makes reading the Perl source code easier. It might also help you interpret the *Devel::Peek* dumps. I'll try to expand it as I learn more.

The first things to look at are the data structures that represent Perl data; scalars of various kinds, arrays and hashes. Internally Perl calls a scalar *SV* (scalar value), an array *AV* (array value) and a hash *HV* (hash value). In addition it uses *IV* for integer value, *NV* for numeric value (aka double), *PV* for a pointer value (aka string value (char*), but 'S' was already taken), and *RV* for reference value. The *IVs* are further guaranteed to be big enough to hold a `void*`.

The internal relationship between the Perl data types is really object oriented. Perl relies on using C's structural equivalence to help emulate something like C++ inheritance of types. The various data types that Perl implement are illustrated in this class hierarchy diagram. The arrows indicate inheritance (IS-A relationships).
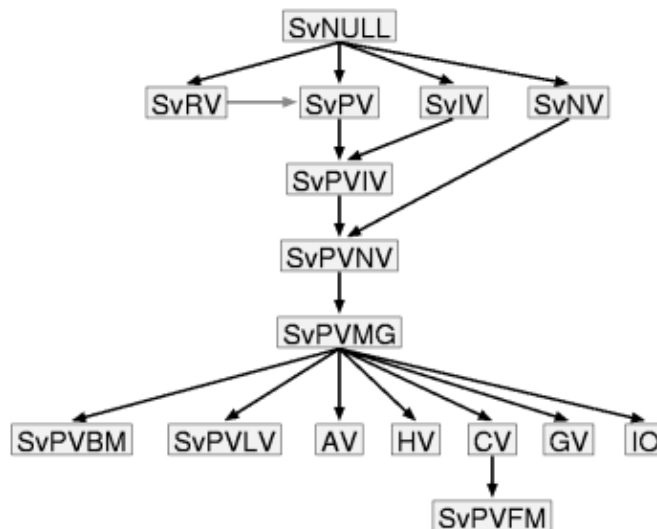
As you can see, Perl uses multiple inheritance with C *SvNULL* (also named just *SV*) acting as some kind of virtual base class. All the Perl types are identified by small numbers, and the internal Perl code often gets away with testing the ISA-relationship between types with the <= operator. As you can see from the figure above, this can only work reliably for some comparisons. All Perl data value objects are tagged with their type, so you can always ask an object what its type is and act according to this information.

The symbolic type names (and associated value) are:

      0) **SVt_NULL**
      1) **SVt_IV**
      2) **SVt_NV**
      3) **SVt_RV**
      4) **SVt_PV**
      5) **SVt_PVIV**

6) **SVt_PVNV**
7) **SVt_PVMG**
8) **SVt_PVBM**
9) **SVt_PVLV**
10) **SVt_PVAV**
11) **SVt_PVHV**
12) **SVt_PVCV**
13) **SVt_PVGV**
14) **SVt_PVFM**
15) **SVt_PVIO**

In addition to the simple type names already mentioned, the following names are found in the hierarchy figure: An *SvPVIV* value can hold a string and an integer value. An *SvPVNV* value can hold a string, an integer and a double value. The *SvPVMG* is used when magic is attached or the value is blessed. The *SvPVBM* adds information for fast searching (Boyer-Moore) on the string value. The *SvPVLV* represents a LValue object. *CV* is a code value, which represents a perl function/subroutine/closure or contains a pointer to an XSUB. *GV* is a glob value and *IO* contains pointers to open files and directories and various state information about these. The *SvPVFM* is used to hold information on forms.

A Perl data object can change type as the value is modified. The SV is said to be upgraded in this case. Type changes only go down the hierarchy. (See the sv_upgrade() function in sv.c.)

The actual layout in memory does not really match how a typical C++ compiler would implement a hierarchy like the one depicted above. Let's see how it is done.

> In the description below we use field names that match the macros that are used to access the corresponding field. For instance the `xpv_cur` field of the `xpvXX` structs are accessed with the `SvCUR()` macro. The field is referred to as **CUR** in the description below. This also match the field names reported by the Devel::Peek module.

SvNULL and struct sv

The simplest type is SvNULL. It always represents an *undefined* scalar value. It consist of the "struct sv" only, and looks like this:
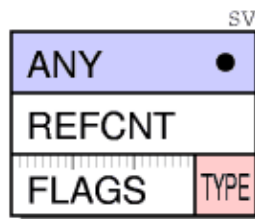

It contains a pointer (ANY) to more data, which in this case is always NULL. All the subtypes are implemented by attaching additional data to the ANY pointer.

The second field is an integer reference counter (REFCNT) which should tell us how many pointers reference this object. When Perl data types are created this value is initialized to 1. The field must be incremented when a new pointer is made to point to it and decremented when the pointer is destroyed or assigned a different value. When the reference count reaches zero the object is freed.

The third word contains a FLAGS field and a TYPE field. The TYPE field contains a small number that represents one of the types shown in the type hierarchy figure above. The FLAGS field has room for 24 flag bits, which encode how various fields of the object should be interpreted, and other state information. Some flags are just used as optimizations in order to avoid having to dereference several levels of pointers just to find that the information is not there.
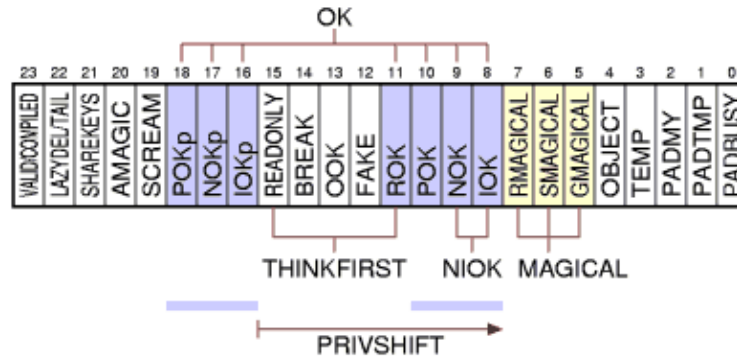
The purpose of the flag bits are:

> 0) **PADBUSY**
>> reserved for tmp or my already
> 1) **PADTMP**
>> in use as tmp
> 2) **PADMY**
>> in use a "my" variable
> 3) **TEMP**
>> string is stealable
> 4) **OBJECT**
>> This flag is set when the object is "blessed". It can only be set for value type SvPVMG or subtypes of it. This flag also indicates that the STASH pointer is valid and points to a namespace HV.
> 5) **GMAGICAL** (Get Magic)
>> This flag indicates that the object has a magic *get* or *len* method to be invoked. It can only be set for value type SvPVMG or subtypes of it. This flag also indicate that the MAGIC pointer is valid.
> 6) **SMAGICAL** (Set Magic)
>> This flag indicates that the object has a magic *set* method to be invoked.
> 7) **RMAGICAL** (Random Magic)
>> This flag indicates that the object has any other magical methods (besides get/len/set magic method) or even methodless magic attached.
>>
>> The RMAGICAL flag is used mainly for tied HV and AV (having 'P' magic) and SVs which have magic *clear* method. It is used as an optimization to avoid setting GMAGICAL and SMAGICAL flags for SVs which need to be marked as MAGICAL otherwise.
>>
>> Any of GMAGICAL, SMAGICAL and RMAGICAL is called MAGICAL
> 8) **IOK** (Integer OK)
>> This flag indicates that the object has a valid IVX field value. It can only be set for value type SvIV or subtypes of it.

9) **NOK** (Numeric OK)

   This flag indicates that the object has a valid NVX field value. It can only be set for value type SvNV or subtypes of it.

10) **POK** (Pointer OK)

   This flag indicates that the object has a valid PVX, CUR and LEN field values (i.e. a valid string value). It can only be set for value type SvPV or subtypes of it.

11) **ROK** (Reference OK)

   This flag indicates that the type should be treated as an SvRV and that the RV field contains a valid reference pointer.

12) **FAKE**

   glob or lexical is just a copy

13) **OOK** (Offset OK)

   This flag indicates that the IVX value is to be interpreted as a string offset. This flag can only be set for value type SvPVIV or subtypes of it. It also follows that the IOK (and IOKp) flag must be off when OOK is on. Take a look at the *SvOOK* figure below.

14) **BREAK**

   refcnt is artificially low

15) **READONLY**

   This flag indicate that the value of the object may not be modified.

16) **IOKp** (Integer OK Private)

   has valid non-public integer value.

   The private OK flags (IOKp, NOKp, POKp) are used by the magic system. During execution of a magic callback, the private flags will be used to set the public flags. When the callback returns, then the public flags are cleared. This effectively is used to pass the value to get/set to/from magic callbacks.

17) **NOKp** (Numeric OK Private)

   has valid non-public numeric value

18) **POKp** (Pointer OK Private)

   has valid non-public pointer value

19) **SCREAM**

   has been studied

20) **AMAGIC**

   has magical overloaded methods

21) **SHAREKEYS**

   Only used by HVs. See description of HV below.

22) **LAZYDEL**

   Only used by HVs. See description of HV below.

22) **TAIL**

   Only used by SvPVBMs. See description of SvPVBM below.

23) **VALID**

   Only used by SvPVBMs. See description of SvPVBM below.

23) **COMPILED**

The `struct sv` is common for all subtypes of SvNULL in Perl. In the Perl source code this structure is typedefed to *SV*, *AV*, *HV*, *CV*, *GV* and *IO*. Routines that can take any type as parameter will have `SV*` as parameter. Routines that only work with arrays or hashes have `AV*` or `HV*` respectively in their parameter list. Likewise for the rest.
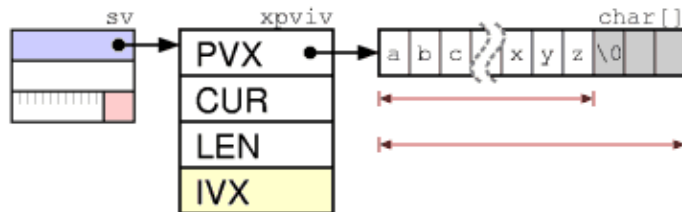
SvPV

A scalar that can hold a string value is called an *SvPV*. In addition to the *SV* struct of SvNULL, an *xpv* struct is allocated and it contains 3 fields. PVX is the pointer to an allocated char array. CUR is an integer giving the current length of the string. LEN is an integer giving the length of the allocated string. The byte at (PVX + CUR) should always be '\0' in order to make sure that the string is NUL-terminated if passed to C library routines. This requires that LEN is always at least 1 larger than CUR.
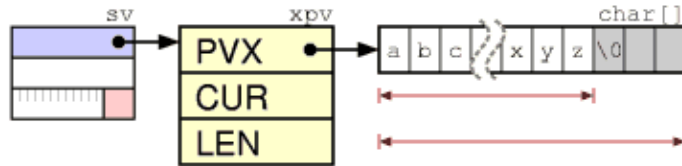

The POK flag indicates that the string pointed to by PVX contains an valid string value. If the POK flag is off and the ROK flag is turned on, then the PVX field is used as a pointer to an SV (see SvRV below) and CUR/LEN is unused. An SvPV with both the POK and ROK flags turned off represents *undef*. The PVX pointer can also be NULL when POK is off and no string storage has been allocated.

SvPVIV and SvPVNV

The *SvPVIV* type is like *SvPV* but has an additional field to hold a single integer value called IVX. The IOK flag indicates if the IVX value is valid. If both the IOK and POK flag is on, then the PVX will (usually) be a string representation of the same number found in IVX.


The *SvPVNV* type is like *SvPVIV* but has an additional field to hold a single *double* value called NVX. The corresponding flag is called NOK.
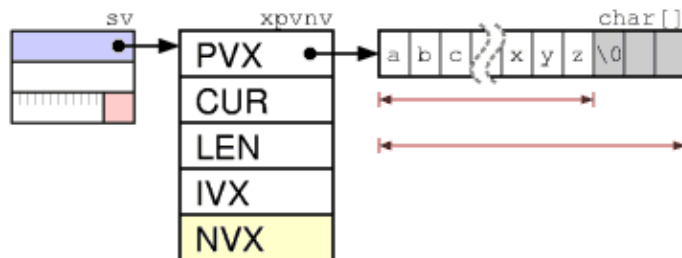

SvOOK

As a special hack, in order to improve the speed of removing characters from the beginning of a string, the OOK flag is used. When this flag is on, then the IVX value is not interpreted as an integer value, but is instead

used as an *offset* into the string. The PVX, CUR, LEN is adjusted to point within the allocated string instead.
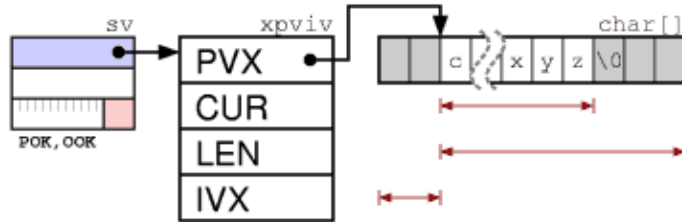

SvIV and SvNV

As a special case we also have *SvIV* and *SvNV* types that only have room for a single integer or a single double value. These are special in that the PVX/CUR/LEN fields are not present even if the ANY pointer actually points to the ghostual incarnation of them. This arrangement makes it possible for code to always access the IVX/NVX fields at a fixed offset from where the SV field ANY points.
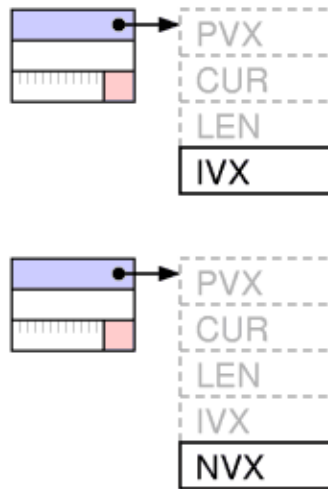

SvRV

The *SvRV* subtype just lets the SV field ANY point to a pointer which points to an SV (which can be any of the SvNULL subtypes). A SvRV object with ROK flag off represents an undefined value.
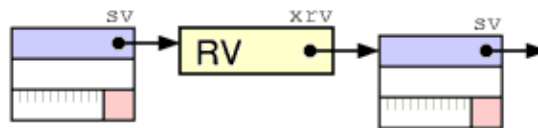
Subclasses of SvPV can also be treated as SvRV objects when the ROK flag is turned on for them. The PVX will then be used as the RV field of the SvRV. Of course ROK and POK is mutually exclusive and the CUR/LEN fields have no meaning when ROK is on.
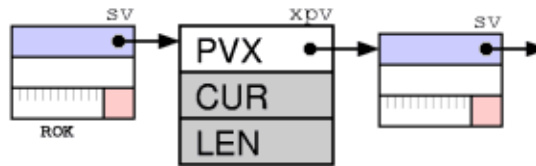
SvPVMG

The *SvPVMG* is like *SvPVNV* above, but has two additional fields; MAGIC and STASH. MAGIC is a pointer to additional structures that contains callback functions and other data. If the MAGIC pointer is non-NULL, then one or more of the MAGICAL flags will be set.

STASH (**s**ymbol **t**able h**ash**) is a pointer to a HV that represents some namespace/class. (That the HV represents some namespace means that the NAME field of the HV must be non-NULL. See description of HVs and stashes below). The STASH field is set when the value is blessed into a package (becomes an object). The OBJECT flag will be set when STASH is. *(IMHO, this field should really have been named "CLASS". The GV and CV subclasses introduce their own unrelated fields called STASH which might be confusing.)*
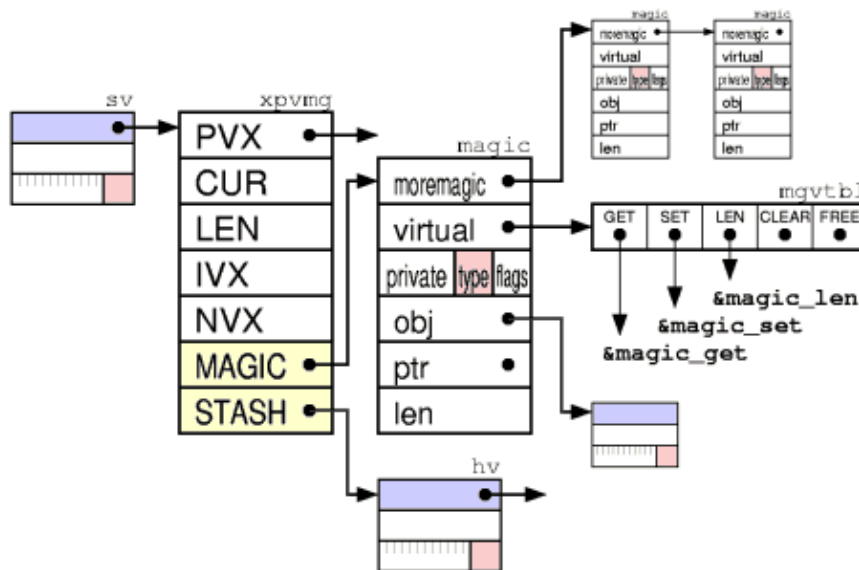
The field MAGIC points to an instance of `struct magic` (typedefed as `MAGIC`). This struct has 8 fields:

1. *moremagic* is a pointer to another MAGIC and is used to form a single linked list of the MAGICs attached to an SV.
2. *virtual* is a pointer to a struct containing 5 function pointers. The functions (if set) are invoked when the corresponding action happens to the SV.
3. *private* is a 16 bit number (U16) not used by Perl.
4. *type* is a character field and is used to denote which kind of magic this is. The interpretation of the rest of the fields depend on the *type* (actually it is the callbacks attached to *virtual* that do any interpretation). There is usually a direct correspondence between the *type* field and the *virtual* field.
5. *flags* contains 8 flag bits. Only 2 are generally used. Bit 1 is the **REFCOUNTED** flag. It indicates that the *obj* is assumed to be an SV and that it's reference count must be decremented when this magic is freed. The **GSKIP** flag indicate that invocation of the magical GET method should be suppressed. Other flag bits are used depending of the kind of magic.

6. *obj* is usually a pointer to some SV. How it is used depends on the kind of magic this is.
7. *ptr* is usually a pointer to some character string. How it is used depends on the kind of magic this is. If the *len* field is >= 0, then *ptr* is assumed to point to a malloced buffer and will be automatically freed when the magic is.
8. *len* is usually the length of the character string pointed to by *ptr*. How it is used depends on the kind of magic this is.

SvPVBM

The *SvPVBM* is like *SvPVMG* above, but has three additional fields; BmUSEFUL, BmPREVIOUS, BmRARE. The SvPVBM value types are used internally to implement very fast lookup of the string in PVX using the Boyer-Moore algorithm. They are used by the Perl index() builtin when the search string is a constant, as well as in the RE engine. The fbm_compile() function turns normal SvPVs into this value type.

A table of 256 elements is appended to the PVX. This table contains the distance from the end of string of the last occurrence of each character in the original string. (In recent Perls, the table is not built for strings shorter than 3 character.) In addition fbm_compile() locates the rarest character in the string (using builtin letter frequency tables) and stores this character in the BmRARE field. The BmPREVIOUS field is set to the location of the first occurrence of the rare character. BmUSEFUL is incremented (decremented) by the RE engine when this constant substring (does not) help in optimizing RE engine access away. If it goes below 0, then the corresponding substring is forgotten and freed;
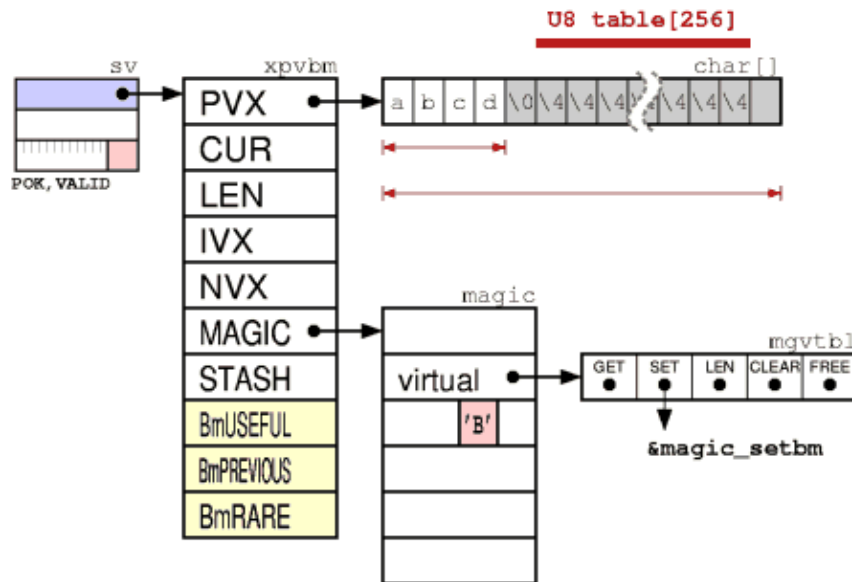
The extra SvPVBM information and the character distance table is only valid when the **VALID** flag is on. A magic structure with the sole purpose of turning off the VALID flag on assignment, is always attached to a *valid* SvPVBM.

The **TAIL** flag is used to indicate that the search for the SvPVMG should be *tail anchored*, i.e. a match should only be considered at the end of the string (or before newline at the end of the string).
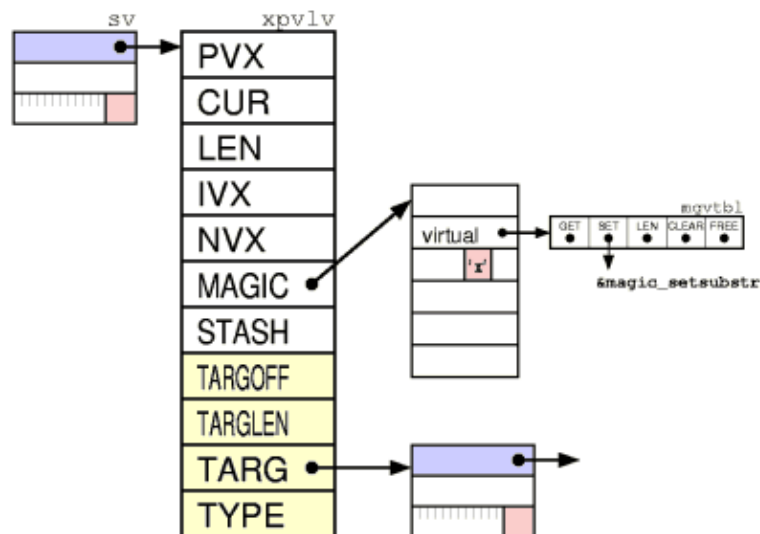
SvPVLV

The *SvPVLV* is like *SvPVMG* above, but has four additional fields; TARGOFF, TARGLEN, TARG, TYPE. The typical use is for Perl builtins that can be used in the LValue context (substr, vec,...). They will return an SvPVLV value, which when assigned to use magic to affect the *target* object, which they keep a pointer to in the TARG field.

The TYPE is a character variable. It encodes the kind if LValue this is. Interpretation of the other LValue fields depend on the TYPE. The SvPVLVs are (almost) always magical. The magic type will match the TYPE field of the SvPVLV. The types are:

'x'

Type-x LVs are returned by the `substr($string, $offset, $len)` builtin.

'v'

Type-v LVs are returned by the `vec($string, $offset, $bits)` builtin.

'.'

Type-. LVs are returned by the `pos($scalar)` builtin.

'k'

Type-k LVs are returned when `keys %hash` is used on the left side of the assignment operator.

'y'

Type-y LVs are used by auto-vivification (of hash and array elements) and the foreach array iterator variable.

'/'

Used by *pp_pushre*. *(I don't understand this yet.)*

The figure below shows an SvPVLV as returned from the `substr()` builtin. The first substr parameter (the string to be affected) is assigned to the TARG field. The substr offset value goes in the TARGOFF field and the substr length parameter goes in the TARGLEN field.


When assignment to an SvPVLV type occurs, then the value to be assigned is first copied into the SvPVLV itself (and affects the PVX, IVX or NVX). After this the magic SET method is invoked, which will update the TARG accordingly.

AV

An array is in many ways represented similar to strings. An AV contains all the fields of SvPVMG and adds the following tree fields: ALLOC is a pointer to the allocated array. ARYLEN is a pointer to a magic SV (which is returned when $#array is requested) and is only created on demand. FLAGS contains some extra flag bits that are specific of the array subtype.

The first three fields of xpvav have been renamed even though they serve nearly the same function as the corresponding SvPV fields. PVX has become ARRAY, CUR has become FILL and LEN has become MAX. One difference is that the value of FILL/MAX is always one less than CUR/LEN would be in the same situation. The IVX/NVX fields are unused.
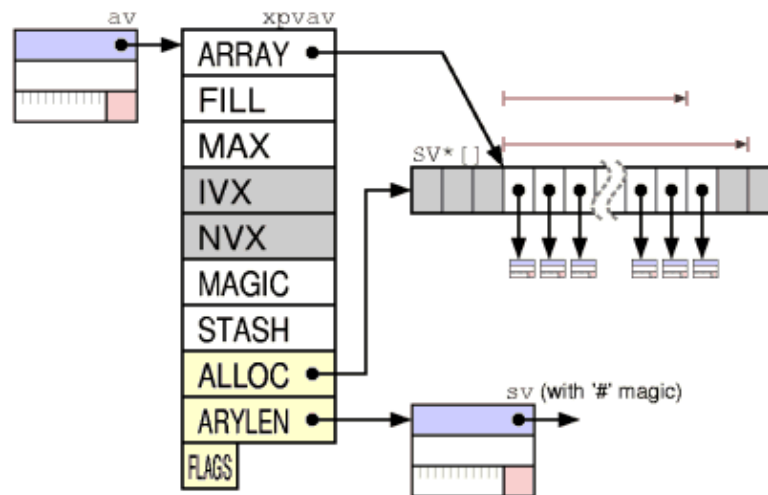
The array pointed to by ARRAY contains pointers to any of the SvNULL subtypes. Usually ALLOC and ARRAY both point to the start of the allocated array. The use of two pointers is similar to the OOK hack described above. The shift operation can be implemented efficiently by just adjusting the ARRAY pointer (and FILL/MAX). Similarly, the pop just involves decrementing the FILL count.

The are only 3 array flags defined:

0) **REAL**

It basically means that all SVs contained in this array is owned and must have their reference counters decremented when the reference is removed from the array. All normal arrays are REAL. For the `stack` the REAL flag is turned off. For `@_` the REAL flag is initially turned off.

1) **REIFY**

The array is *not* REAL but should be made REAL if modified. The `@_` array will have the REIFY flag turned on.

2) **REUSED**

This flag does not seem to be used for anything currently.

HV

Hashes are the most complex of the Perl data types. In addition to what we have seen above, HVs use *HE* structs to represent key/value pairs and *HEK* structs to represent keys.

The HV type itself contains all the fields of SvPVMG and then adds four new fields:

RITER, EITER:

> The first two fields are used to implement a single iterator over the elements in the hash. RITER which is an integer index into the array referenced by ARRAY and EITER which is a pointer to an HE. In order find the next hash element one would first look at EITER->next and if it turns out to be NULL, RITER is incremented until ARRAY[RITER] is non-NULL. The iterator starts out with RITER = -1 and EITER = NULL.

RMROOT/NAME:

> The last two fields are only used when the hash represents a name space (*stash*). PMROOT points to a node in the Perl syntax tree. It is used to implement the reset() builtin for REs. NAME is a NUL-terminated string which denotes the fully qualified name of the name space (aka *package*). This is one of the few places where Perl does not allow strings with embedded NULs.

The first few fields of the xpvhv have been renamed in the same way as for AVs. MAX is the number of elements in ARRAY minus one. (The size of the ARRAY is required to be a power of 2, since the code that deals with hashes just mask off the last few bits of the HASH value to locate the correct HE column for a key: `ARRAY[HASH & MAX]`). Also note that ARRAY can be NULL when the hash is empty (but the MAX value will still be at least 7, which is the minimum value assigned by Perl.) The FILL is the number of elements in ARRAY which are not NULL. The IVX field has been renamed KEYS and is the number of hash elements in the HASH. The NVX field is unused.

The **HE**s are simple structs containing 3 pointers. A pointer to the next HE, a pointer to the key and a pointer to the value of the given hash element.

The **HEK**s are special variable sized structures that store the hash keys. They contain 3 fields. The computed *hash* value of the string, the *len*gth of the string, and *len*+1 bytes for the key string itself (including trailing NUL). As a special case, a *len* value of `HEf_SVKEY` (-2) indicate that a pointer to an SV is stored in the HEK instead of a string. This hack is used for some magical hashes.
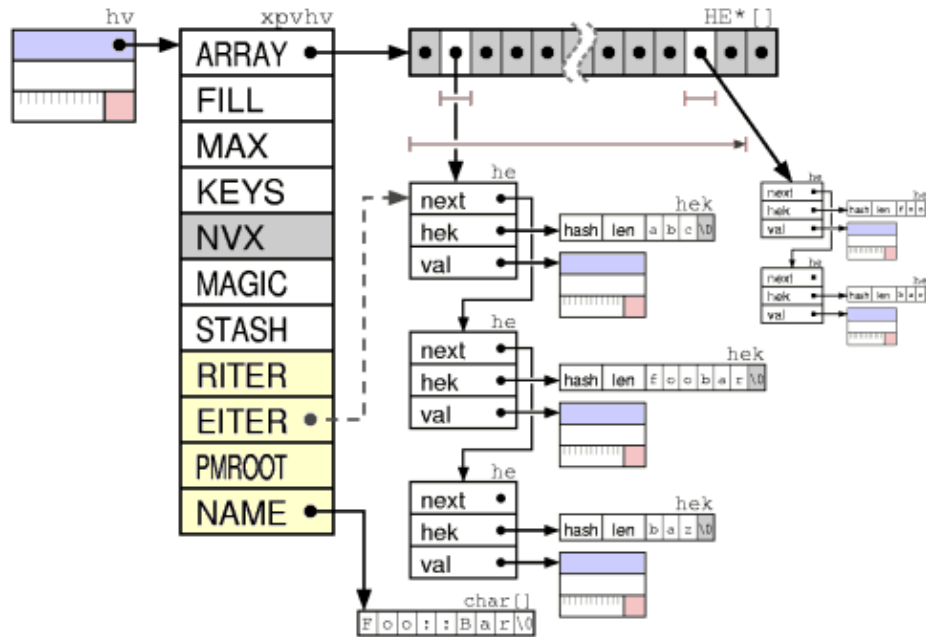
In a perfect hash both KEYS and FILL are the same value. This means than all HEs can be located directly from the pointer in the ARRAY (and all the he->next pointers are NULL).

The following two hash specific flags are found among the common SvNULL flags:

21) **SHAREKEYS**

When this flag is set, then the hash will share the HEK structures with a special hash pointed to by the `strtab` variable. This reduce the storage occupied by hash keys, especially when we have lots of hashes with the same keys. The SHAREKEYS flag is on by default for newly created HVs.
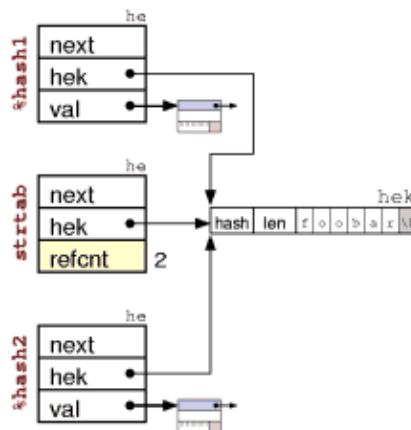
What is special with the `strtab` hash is that the *val* field of the HE structs is used as a reference counter for the HEK. The counter is incremented when new hashes link up this HEK and decremented when the key is removed from the hashes. When the reference count reach 0, the HEK (and corresponding HE) is removed from `strtab` and the storage is freed.

22) **LAZYDEL**

This flag indicates that the hash element pointed to by EITER is really deleted. When you delete the current hash element, perl only marks the HV with the LAZYDEL flag, and when the iterator is advanced, then the element is zapped. This makes it possible to delete elements in a hash while iterating over it.

GV

GV ("glob value" aka "symbol") is like *SvPVMG* above, but has five additional fields; GP, NAME, NAMELEN, GvSTASH, FLAGS.

The GP is a pointer to structure that holds pointers to data of various kinds. Perl use a pointer, instead of including the GP fields in the xpvgv, in order to implement the proper glob aliasing behavior (i.e. different GVs can share the same GP).

The NAME/NAMELEN denotes the unqualified name of this symbol and GvSTASH points to the symbol table where this symbol belongs. The fully qualified symbol name is obtained by taking the NAME of the GvSTASH (see HV above) and appending "::" and NAME to it. The hash pointed to by GvSTASH will usually contain an element with NAME as key and a pointer to this GV as value. See description of stashes below.

The FLAGS field contains space for 8 additional flag bits:

> 0) **INTRO**
> 1) **MULTI**
>> Have we seen more than one occurrence of this glob. Used to implement the "possibly typo" warning.
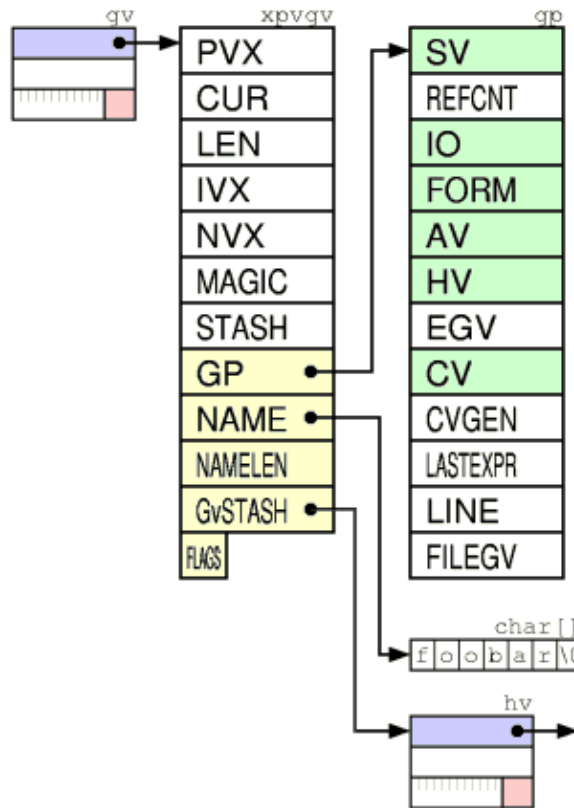> 2) **ASSUMECV**
> 4) **IMPORTED_SV**
> 5) **IMPORTED_AV**
> 6) **IMPORTED_HV**
> 7) **IMPORTED_CV**

A magic of type '*' is always attached to the GV (not shown in the figure). The magic GET method is used to stringify the globs (as the fully qualified name prefixed with '*'). The magic SET method is used to alias an GLOB based on the name of another glob.

GP

GPs can be shared between one or more GVs. The data type fields for the GP are: SV, IO, FORM, AV, HV, CV. These hold a pointer to the corresponding data type object. (The SV must point to some simple SvNULL subtype (i.e. with type <= SVt_PVLV). The FORM field must point to a SvPVFM if non-NULL. The IO field must point to an IO if non-NULL, the AV to an AV, etc.) The SV is always present (but might point to a SvNULL object). All the others are initially NULL.

The additional administrative fields in the GP are: REFCNT, EGV, CVGEN, LASTEXPR, LINE, FILEGV.

REFCNT is a reference counter. It says how many GVs have a pointer to this GP. It is incremented/decremented as new GVs reference/forget this GP. When the counter reach 0 the GP is freed.

EGV is a pointer to the GV that originally created this GP (used to tell the real name of any aliased symbol). If the original GV is freed, but GP should stay since another GV reference it, then the EGV is NULLed.

CVGEN is an integer used to validate method cache CV entries in the GP. If CVGEN is zero, then the CV is real. If CVGEN is non-zero, but less than the global variable `subgeneration`, then the CV contains a stale method cache entry. If CVGEN is equal to `subgeneration` then the CV contains a valid method cache entry. (Every time some operation that might invalidate some of the method caches are performed, then the `subgeneration` variable is incremented.)

LASTEXPR is an integer I don't understand the purpose of yet.

FILEGV is a pointer to a GV where the SV of the GP is the name of the file where this symbol was first created. The NAME of this GV is the filename prefixed with "_<" and GvSTASH is `defstash` (see

description of stashes below).

LINE is the corresponding line number in the file.

Stashes

GVs and stashes work together to implement the name spaces of Perl. Stashes are named HVs with all the element values being pointers to GVs. The root of the namespace is pointed to by the global variable `defstash`.

In the figure below we have simplified the representation of stashes to a single box. The text in the blue field is the NAME of the HV/stash. The hash elements keys are shown as field names and the element values are shown as a pointers to globs (GV). The GVs are also simplified to a single box. The text in the green field in the fully qualified name of the GV. Only the GP data fields are shown (and FORM has been eliminated because it was not 2 letters long :-).

The figure illustrates how the scalar variables `$::foo` and `$foo::bar::baz` are represented by Perl.
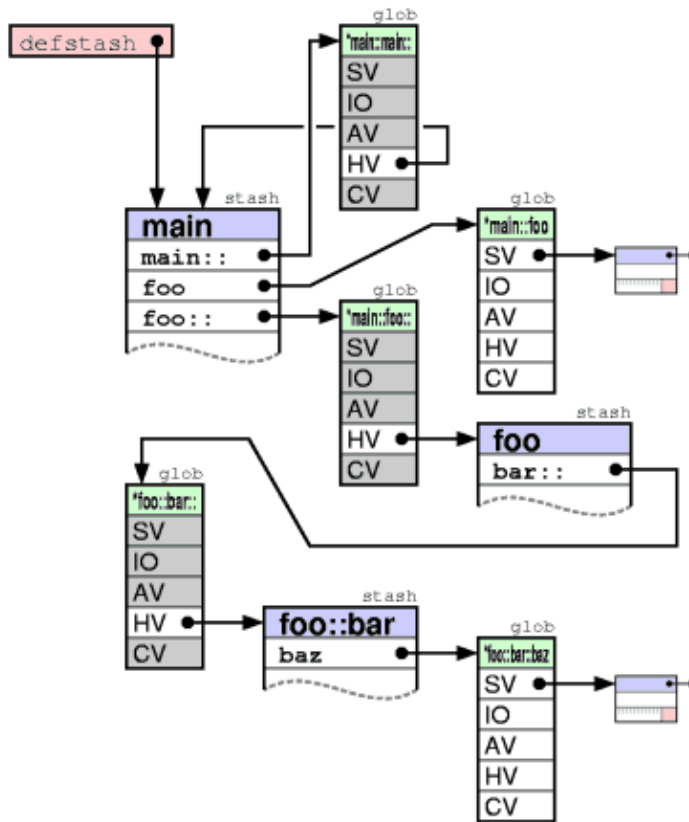
All resolution of qualified names starts with the stash pointed to by the `defstash` variable. Nested name spaces are implemented by a stash entry with a key ending in "`::`". The entry for "`main::`" ensures that `defstash` is also known as "`main`" package (and has the side-effect that the "`main::main::main`" package is `defstash` too.) Unqualified names are resolved starting at `curstash` or `curcop->cop_stash` which are influenced by the `package` declaration in Perl.

As you can see from this figure, there are lots of pointers to dereference in order to look up deeply nested names. Each stash is at least 4 levels deep and each glob is 3 levels, giving at least 24 pointer dereferences to

access the data in the `$foo::bar::baz` variable from `defstash`.

The `defstash` stash is also a place where globs representing source files are entered. These entries are prefixed with "_<". The FILEGV field of the GP points to the same glob as the corresponding "_<" entry in `defstash` does.

CV

The *CV* ("code value") is like *SvPVMG* above, but has thirteen additional fields; CvSTASH, START, ROOT, XSUB, XSUBANY, GV, FILEGV, DEPTH, PADLIST, OUTSIDE, MUTEXP, OWNER, FLAGS. (The fields MUTEXP and OWNER are only present for Perls that support threading).
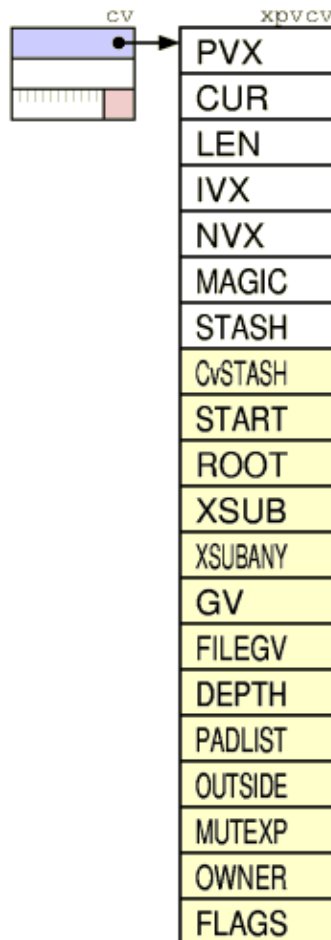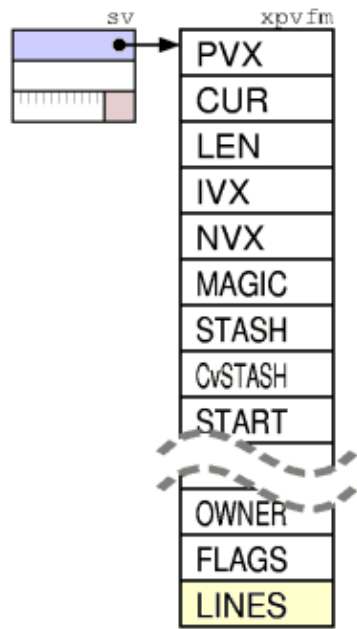

I will need to describe PADs and OPs before detailed description of CVs makes sense.

SvPVFM

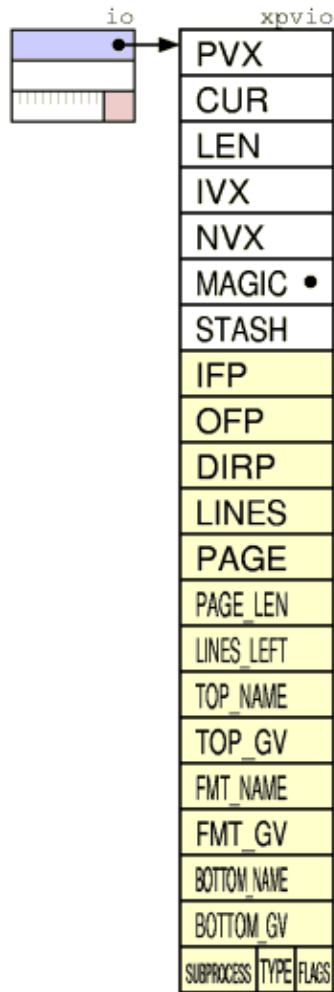The *SvPVFM* is like *CV* above, but adds a single field called LINES.

IO

The *IO* is like *SvPVMG* above, but has quite a few additional fields.

PAD

NOTYET

OP

I will perhaps make a separate document that describes <u>syntax trees and OP codes</u>.
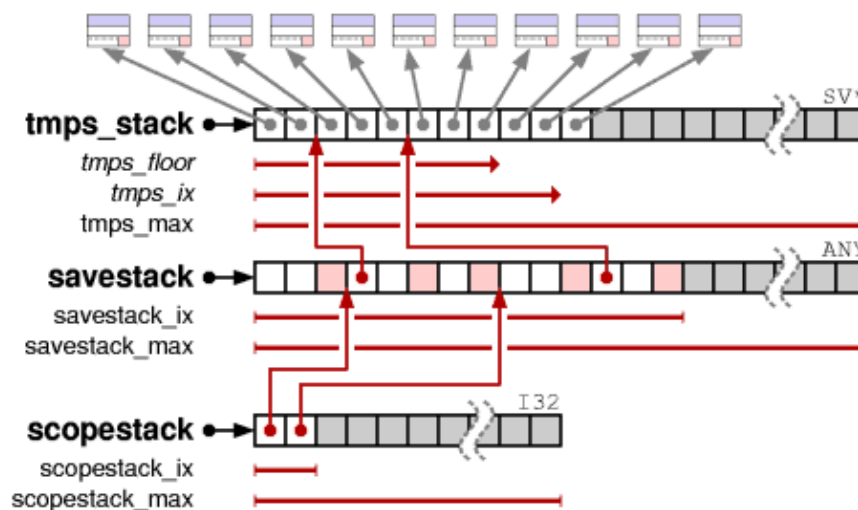
Stacks

During compilation and runtime Perl use several stacks to manage itself and the program running. The first three described here implements scopes, including variables and values which are restored (or actions to be performed) when the scope is left.

The `scopestack` pushes the `savestack_ix` when `ENTER` is executed. On `LEAVE` the top `savestack_ix` entry is popped and all things saved on the `savestack` since this is restored. This means that a `ENTER/LEAVE` pairs represents dynamic nestable scopes.

The `savestack` contains records of things saved in order to be restored when the scopes are left. Each record consist of 2-4 ANY elements. The first one is a type code, which is used to decide how long the record is and how to interpret the other elements. (In the figure the type codes are marked pinkish color.) The restoring involves updating memory locations of various types as well as more general callbacks (destructors).

The `tmps_stack` implement mortal SVs. Each time a new mortal is made, then `tmps_ix` is incremented and the corresponding entry in `tmps_stack` made to point to it. When SAVETMPS is executed, then the old `tmps_floor` value is saved on the `savestack and then tmps_floor is set equal to tmps_ix. When FREETMPS is executed, then all SVs pointed to by the pointers between tmps_floor and tmps_ix will have their REFCNT decremented. How many this will be depend on how many scopes has been left. Note that the tmps_floor and tmps_ix values is the index of the last SV* pushed. They both start out as -1 when the stack is empty.`

The next fours stacks manage subroutine calls, evals, and loops.

The first one is simply denoted as *the stack* and is really an AV. The variable `curstack` points to this AV. To speed up access Perl also maintain direct pointers to the start (`stack_base`) and the end (`stack_max`) of the allocated ARRAY of this AV. The AV so special that it is marked as not REAL and the FILL field is not updated. Instead we use a dedicated pointed called `stack_sp`, the stack pointer. The stack is used to pass arguments to PP operations and subroutines and is also the place where the result of these operations as well as subroutine return values are placed.

The `markstack` is used to indicate the extent of the stack to be passed as @_ to Perl subroutines. When a subroutine is to be called, then first the start of the arguments are marked by pushing the `stack_sp` offset onto `markstack`, then the arguments themselves are calculated and pushed on the stack. Then the @_ array is set up with pointers the SV* on the stack between the MARK and `stack_sp` and the subroutine starts running. For XSUB routines, the creation of @_ is suppressed, and the routine will use the MARK directly to find it's arguments.

The `retstack` contains pointers to the operation to go to after subroutines return. Each time a subroutine is called a new OP* is pushed on this stack. When a subroutine returns, Perl pop the top OP* from `retstack` and continue execution from this place.

The cxstack (context stack) contains records that describe the current context. Each time a subroutine, an eval or a loop is entered, then a new PERL_CONTEXT record is pushed on the cxstack. When subroutine/eval/loop finished then the record top record is poped and the corresponding values restored. The PERL_CONTEXT record will eventually be descibed below.
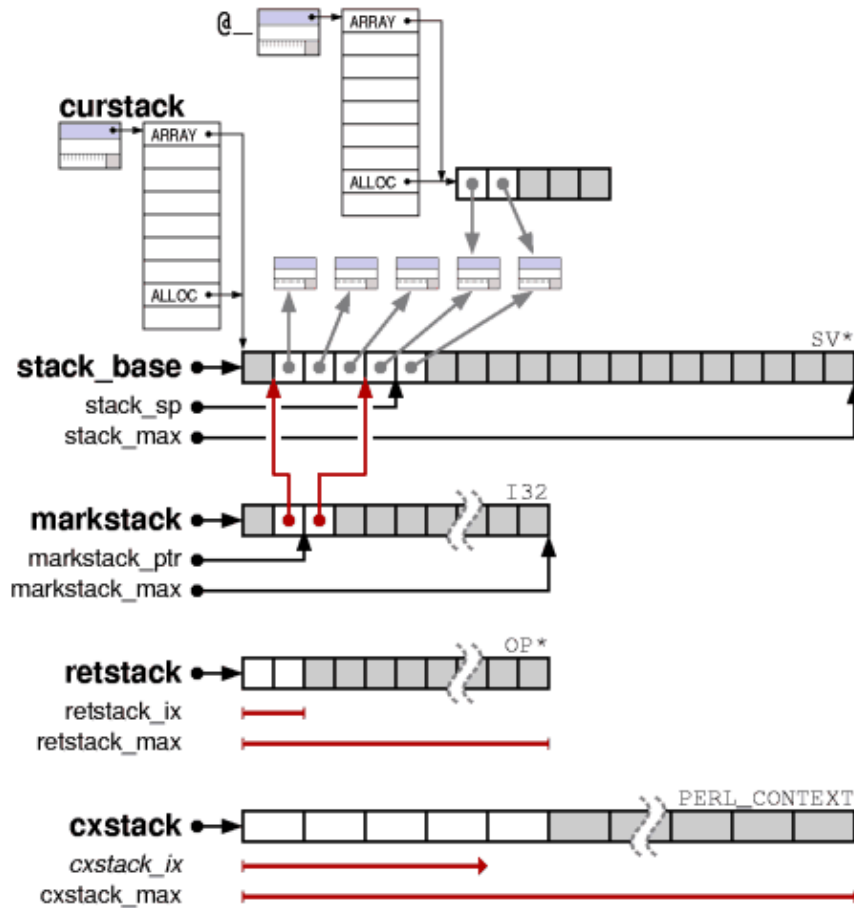
PERL_CONTEXT

NOTYET

REGEXP

NOTYET

PerlInterpreter

NOTYET