NAME

perlguts − Perl's Internal Functions

DESCRIPTION

This document attempts to describe some of the internal functions of the Perl executable. It is far from complete and probably contains many errors. Please refer any questions or comments to the author below.

Variables

Datatypes

Perl has three typedefs that handle Perl's three main data types:

```
SV  Scalar Value
AV  Array Value
HV  Hash Value
```

Each typedef has specific routines that manipulate the various data types.

What is an "IV"?

Perl uses a special typedef IV which is a simple integer type that is guaranteed to be large enough to hold a pointer (as well as an integer).

Perl also uses two special typedefs, I32 and I16, which will always be at least 32-bits and 16-bits long, respectively.

Working with SVs

An SV can be created and loaded with one command. There are four types of values that can be loaded: an integer value (IV), a double (NV), a string, (PV), and another scalar (SV).

The six routines are:

```
SV*  newSViv(IV);
SV*  newSVnv(double);
SV*  newSVpv(char*, int);
SV*  newSVpvn(char*, int);
SV*  newSVpvf(const char*, ...);
SV*  newSVsv(SV*);
```

To change the value of an *already-existing* SV, there are seven routines:

```
void  sv_setiv(SV*, IV);
void  sv_setuv(SV*, UV);
void  sv_setnv(SV*, double);
void  sv_setpv(SV*, const char*);
void  sv_setpvn(SV*, const char*, int)
void  sv_setpvf(SV*, const char*, ...);
void  sv_setpvfn(SV*, const char*, STRLEN, va_list *, SV **, I32, bool);
void  sv_setsv(SV*, SV*);
```

Notice that you can choose to specify the length of the string to be assigned by using `sv_setpvn`, `newSVpvn`, or `newSVpv`, or you may allow Perl to calculate the length by using `sv_setpv` or by specifying 0 as the second argument to `newSVpv`. Be warned, though, that Perl will determine the string's length by using `strlen`, which depends on the string terminating with a NUL character.

The arguments of `sv_setpvf` are processed like `sprintf`, and the formatted output becomes the value.

`sv_setpvfn` is an analogue of `vsprintf`, but it allows you to specify either a pointer to a variable argument list or the address and length of an array of SVs. The last argument points to a boolean; on return, if that boolean is true, then locale-specific information has been used to format the string, and the string's contents are therefore untrustworthy (see the *perlsec* manpage). This pointer may be NULL if that information is not important. Note that this function requires you to specify the length of the format.

The `sv_set*()` functions are not generic enough to operate on values that have ‘‘magic’’. See the section on *Magic Virtual Tables* later in this document.

All SVs that contain strings should be terminated with a NUL character. If it is not NUL−terminated there is a risk of core dumps and corruptions from code which passes the string to C functions or system calls which expect a NUL−terminated string. Perl’s own functions typically add a trailing NUL for this reason. Nevertheless, you should be very careful when you pass a string stored in an SV to a C function or system call.

To access the actual value that an SV points to, you can use the macros:

```
SvIV(SV*)
SvNV(SV*)
SvPV(SV*, STRLEN len)
```

which will automatically coerce the actual scalar type into an IV, double, or string.

In the `SvPV` macro, the length of the string returned is placed into the variable `len` (this is a macro, so you do *not* use `&len`). If you do not care what the length of the data is, use the global variable `PL_na` or a local variable of type `STRLEN`. However using `PL_na` can be quite inefficient because `PL_na` must be accessed in thread-local storage in threaded Perl. In any case, remember that Perl allows arbitrary strings of data that may both contain NULs and might not be terminated by a NUL.

Also remember that C doesn’t allow you to safely say `foo(SvPV(s, len), len);`. It might work with your compiler, but it won’t work for everyone. Break this sort of statement up into separate assignments:

```
        STRLEN len;
        char * ptr;
        ptr = SvPV(len);
        foo(ptr, len);
```

If you want to know if the scalar value is TRUE, you can use:

```
SvTRUE(SV*)
```

Although Perl will automatically grow strings for you, if you need to force Perl to allocate more memory for your SV, you can use the macro

```
SvGROW(SV*, STRLEN newlen)
```

which will determine if more memory needs to be allocated. If so, it will call the function `sv_grow`. Note that `SvGROW` can only increase, not decrease, the allocated memory of an SV and that it does not automatically add a byte for the a trailing NUL (perl’s own string functions typically do `SvGROW(sv, len + 1)`).

If you have an SV and want to know what kind of data Perl thinks is stored in it, you can use the following macros to check the type of SV you have.

```
SvIOK(SV*)
SvNOK(SV*)
SvPOK(SV*)
```

You can get and set the current length of the string stored in an SV with the following macros:

```
SvCUR(SV*)
SvCUR_set(SV*, I32 val)
```

You can also get a pointer to the end of the string stored in the SV with the macro:

```
SvEND(SV*)
```

But note that these last three macros are valid only if `SvPOK()` is true.

If you want to append something to the end of string stored in an `SV*`, you can use the following functions:

```
void  sv_catpv(SV*, char*);
void  sv_catpvn(SV*, char*, STRLEN);
void  sv_catpvf(SV*, const char*, ...);
void  sv_catpvfn(SV*, const char*, STRLEN, va_list *, SV **, I32, bool);
void  sv_catsv(SV*, SV*);
```

The first function calculates the length of the string to be appended by using `strlen`. In the second, you specify the length of the string yourself. The third function processes its arguments like `sprintf` and appends the formatted output. The fourth function works like `vsprintf`. You can specify the address and length of an array of SVs instead of the va_list argument. The fifth function extends the string stored in the first SV with the string stored in the second SV. It also forces the second SV to be interpreted as a string.

The `sv_cat*()` functions are not generic enough to operate on values that have ''magic''. See the section on *Magic Virtual Tables* later in this document.

If you know the name of a scalar variable, you can get a pointer to its SV by using the following:

```
SV*  perl_get_sv("package::varname", FALSE);
```

This returns NULL if the variable does not exist.

If you want to know if this variable (or any other SV) is actually `defined`, you can call:

```
SvOK(SV*)
```

The scalar `undef` value is stored in an SV instance called `PL_sv_undef`. Its address can be used whenever an `SV*` is needed.

There are also the two values `PL_sv_yes` and `PL_sv_no`, which contain Boolean TRUE and FALSE values, respectively. Like `PL_sv_undef`, their addresses can be used whenever an `SV*` is needed.

Do not be fooled into thinking that `(SV *) 0` is the same as `&PL_sv_undef`. Take this code:

```
SV* sv = (SV*) 0;
if (I-am-to-return-a-real-value) {
        sv = sv_2mortal(newSViv(42));
}
sv_setsv(ST(0), sv);
```

This code tries to return a new SV (which contains the value 42) if it should return a real value, or undef otherwise. Instead it has returned a NULL pointer which, somewhere down the line, will cause a segmentation violation, bus error, or just weird results. Change the zero to `&PL_sv_undef` in the first line and all will be well.

To free an SV that you've created, call `SvREFCNT_dec(SV*)`. Normally this call is not necessary (see the section on *Reference Counts and Mortality*).

What's Really Stored in an SV?

Recall that the usual method of determining the type of scalar you have is to use `Sv*OK` macros. Because a scalar can be both a number and a string, usually these macros will always return TRUE and calling the `Sv*V` macros will do the appropriate conversion of string to integer/double or integer/double to string.

If you *really* need to know if you have an integer, double, or string pointer in an SV, you can use the following three macros instead:

```
SvIOKp(SV*)
SvNOKp(SV*)
SvPOKp(SV*)
```

These will tell you if you truly have an integer, double, or string pointer stored in your SV. The ‘‘p’’ stands for private.

In general, though, it’s best to use the `Sv*V` macros.

Working with AVs

There are two ways to create and load an AV. The first method creates an empty AV:

```
AV*  newAV();
```

The second method both creates the AV and initially populates it with SVs:

```
AV*  av_make(I32 num, SV **ptr);
```

The second argument points to an array containing `num` `SV*`’s. Once the AV has been created, the SVs can be destroyed, if so desired.

Once the AV has been created, the following operations are possible on AVs:

```
void  av_push(AV*, SV*);
SV*   av_pop(AV*);
SV*   av_shift(AV*);
void  av_unshift(AV*, I32 num);
```

These should be familiar operations, with the exception of `av_unshift`. This routine adds `num` elements at the front of the array with the `undef` value. You must then use `av_store` (described below) to assign values to these new elements.

Here are some other functions:

```
I32   av_len(AV*);
SV**  av_fetch(AV*, I32 key, I32 lval);
SV**  av_store(AV*, I32 key, SV* val);
```

The `av_len` function returns the highest index value in array (just like $#array in Perl). If the array is empty, −1 is returned. The `av_fetch` function returns the value at index `key`, but if `lval` is non-zero, then `av_fetch` will store an undef value at that index. The `av_store` function stores the value `val` at index `key`, and does not increment the reference count of `val`. Thus the caller is responsible for taking care of that, and if `av_store` returns NULL, the caller will have to decrement the reference count to avoid a memory leak. Note that `av_fetch` and `av_store` both return `SV**`’s, not `SV*`’s as their return value.

```
void  av_clear(AV*);
void  av_undef(AV*);
void  av_extend(AV*, I32 key);
```

The `av_clear` function deletes all the elements in the AV* array, but does not actually delete the array itself. The `av_undef` function will delete all the elements in the array plus the array itself. The `av_extend` function extends the array so that it contains at least `key+1` elements. If `key+1` is less than the currently allocated length of the array, then nothing is done.

If you know the name of an array variable, you can get a pointer to its AV by using the following:

```
AV*  perl_get_av("package::varname", FALSE);
```

This returns NULL if the variable does not exist.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use the array access functions on tied arrays.

Working with HVs

To create an HV, you use the following routine:

```
HV*  newHV();
```

Once the HV has been created, the following operations are possible on HVs:

```
SV**  hv_store(HV*, char* key, U32 klen, SV* val, U32 hash);
SV**  hv_fetch(HV*, char* key, U32 klen, I32 lval);
```

The `klen` parameter is the length of the key being passed in (Note that you cannot pass 0 in as a value of `klen` to tell Perl to measure the length of the key). The `val` argument contains the SV pointer to the scalar being stored, and `hash` is the precomputed hash value (zero if you want `hv_store` to calculate it for you). The `lval` parameter indicates whether this fetch is actually a part of a store operation, in which case a new undefined value will be added to the HV with the supplied key and `hv_fetch` will return as if the value had already existed.

Remember that `hv_store` and `hv_fetch` return `SV**`'s and not just `SV*`. To access the scalar value, you must first dereference the return value. However, you should check to make sure that the return value is not NULL before dereferencing it.

These two functions check if a hash table entry exists, and deletes it.

```
bool  hv_exists(HV*, char* key, U32 klen);
SV*   hv_delete(HV*, char* key, U32 klen, I32 flags);
```

If `flags` does not include the `G_DISCARD` flag then `hv_delete` will create and return a mortal copy of the deleted value.

And more miscellaneous functions:

```
void   hv_clear(HV*);
void   hv_undef(HV*);
```

Like their AV counterparts, `hv_clear` deletes all the entries in the hash table but does not actually delete the hash table. The `hv_undef` deletes both the entries and the hash table itself.

Perl keeps the actual data in linked list of structures with a typedef of HE. These contain the actual key and value pointers (plus extra administrative overhead). The key is a string pointer; the value is an `SV*`. However, once you have an `HE*`, to get the actual key and value, use the routines specified below.

```
I32    hv_iterinit(HV*);
        /* Prepares starting point to traverse hash table */
HE*    hv_iternext(HV*);
        /* Get the next entry, and return a pointer to a
           structure that has both the key and value */
char*  hv_iterkey(HE* entry, I32* retlen);
        /* Get the key from an HE structure and also return
           the length of the key string */
SV*    hv_iterval(HV*, HE* entry);
        /* Return a SV pointer to the value of the HE
           structure */
SV*    hv_iternextsv(HV*, char** key, I32* retlen);
        /* This convenience routine combines hv_iternext,
           hv_iterkey, and hv_iterval.  The key and retlen
           arguments are return values for the key and its
           length.  The value is returned in the SV* argument */
```

If you know the name of a hash variable, you can get a pointer to its HV by using the following:

```
HV*  perl_get_hv("package::varname", FALSE);
```

This returns NULL if the variable does not exist.

The hash algorithm is defined in the `PERL_HASH(hash, key, klen)` macro:

```
hash = 0;
while (klen--)
    hash = (hash * 33) + *key++;
```

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use the hash access functions on tied hashes.

Hash API Extensions

Beginning with version 5.004, the following functions are also supported:

```
HE*     hv_fetch_ent  (HV* tb, SV* key, I32 lval, U32 hash);
HE*     hv_store_ent  (HV* tb, SV* key, SV* val, U32 hash);

bool    hv_exists_ent (HV* tb, SV* key, U32 hash);
SV*     hv_delete_ent (HV* tb, SV* key, I32 flags, U32 hash);

SV*     hv_iterkeysv  (HE* entry);
```

Note that these functions take `SV*` keys, which simplifies writing of extension code that deals with hash structures. These functions also allow passing of `SV*` keys to `tie` functions without forcing you to stringify the keys (unlike the previous set of functions).

They also return and accept whole hash entries (`HE*`), making their use more efficient (since the hash number for a particular string doesn't have to be recomputed every time). See the section on *API LISTING* later in this document for detailed descriptions.

The following macros must always be used to access the contents of hash entries. Note that the arguments to these macros must be simple variables, since they may get evaluated more than once. See the section on *API LISTING* later in this document for detailed descriptions of these macros.

```
HePV(HE* he, STRLEN len)
HeVAL(HE* he)
HeHASH(HE* he)
HeSVKEY(HE* he)
HeSVKEY_force(HE* he)
HeSVKEY_set(HE* he, SV* sv)
```

These two lower level macros are defined, but must only be used when dealing with keys that are not `SV*`s:

```
HeKEY(HE* he)
HeKLEN(HE* he)
```

Note that both `hv_store` and `hv_store_ent` do not increment the reference count of the stored `val`, which is the caller's responsibility. If these functions return a NULL value, the caller will usually have to decrement the reference count of `val` to avoid a memory leak.

References

References are a special type of scalar that point to other data types (including references).

To create a reference, use either of the following functions:

```
SV* newRV_inc((SV*) thing);
SV* newRV_noinc((SV*) thing);
```

The `thing` argument can be any of an `SV*`, `AV*`, or `HV*`. The functions are identical except that `newRV_inc` increments the reference count of the `thing`, while `newRV_noinc` does not. For historical reasons, `newRV` is a synonym for `newRV_inc`.

Once you have a reference, you can use the following macro to dereference the reference:

```
SvRV(SV*)
```

then call the appropriate routines, casting the returned `SV*` to either an `AV*` or `HV*`, if required.

To determine if an SV is a reference, you can use the following macro:

```
SvROK(SV*)
```

To discover what type of value the reference refers to, use the following macro and then check the return value.

```
SvTYPE(SvRV(SV*))
```

The most useful types that will be returned are:

```
SVt_IV    Scalar
SVt_NV    Scalar
SVt_PV    Scalar
SVt_RV    Scalar
SVt_PVAV  Array
SVt_PVHV  Hash
SVt_PVCV  Code
SVt_PVGV  Glob (possible a file handle)
SVt_PVMG  Blessed or Magical Scalar

See the sv.h header file for more details.
```

Blessed References and Class Objects

References are also used to support object-oriented programming. In the OO lexicon, an object is simply a reference that has been blessed into a package (or class). Once blessed, the programmer may now use the reference to access the various methods in the class.

A reference can be blessed into a package with the following function:

```
SV* sv_bless(SV* sv, HV* stash);
```

The `sv` argument must be a reference. The `stash` argument specifies which class the reference will belong to. See the section on *Stashes and Globs* for information on converting class names into stashes.

/* Still under construction */

Upgrades rv to reference if not already one. Creates new SV for rv to point to. If `classname` is non-null, the SV is blessed into the specified class. SV is returned.

```
    SV* newSVrv(SV* rv, char* classname);
```

Copies integer or double into an SV whose reference is `rv`. SV is blessed if `classname` is non-null.

```
    SV* sv_setref_iv(SV* rv, char* classname, IV iv);
    SV* sv_setref_nv(SV* rv, char* classname, NV iv);
```

Copies the pointer value (*the address, not the string!*) into an SV whose reference is rv. SV is blessed if `classname` is non-null.

```
SV* sv_setref_pv(SV* rv, char* classname, PV iv);
```

Copies string into an SV whose reference is `rv`. Set length to 0 to let Perl calculate the string length. SV is blessed if `classname` is non-null.

```
SV* sv_setref_pvn(SV* rv, char* classname, PV iv, STRLEN length);
```

Tests whether the SV is blessed into the specified class. It does not check inheritance relationships.

```
int  sv_isa(SV* sv, char* name);
```

Tests whether the SV is a reference to a blessed object.

```
int  sv_isobject(SV* sv);
```

Tests whether the SV is derived from the specified class. SV can be either a reference to a blessed object or a string containing a class name. This is the function implementing the `UNIVERSAL::isa` functionality.

```
bool sv_derived_from(SV* sv, char* name);
```

To check if you've got an object derived from a specific class you have to write:

```
if (sv_isobject(sv) && sv_derived_from(sv, class)) { ... }
```

Creating New Variables

To create a new Perl variable with an undef value which can be accessed from your Perl script, use the following routines, depending on the variable type.

```
SV*  perl_get_sv("package::varname", TRUE);
AV*  perl_get_av("package::varname", TRUE);
HV*  perl_get_hv("package::varname", TRUE);
```

Notice the use of TRUE as the second parameter. The new variable can now be set, using the routines appropriate to the data type.

There are additional macros whose values may be bitwise OR'ed with the `TRUE` argument to enable certain extra features. Those bits are:

```
GV_ADDMULTI Marks the variable as multiply defined, thus preventing the
            "Name <varname> used only once: possible typo" warning.
GV_ADDWARN  Issues the warning "Had to create <varname> unexpectedly" if
            the variable did not exist before the function was called.
```

If you do not specify a package name, the variable is created in the current package.

Reference Counts and Mortality

Perl uses an reference count-driven garbage collection mechanism. SVs, AVs, or HVs (xV for short in the following) start their life with a reference count of 1. If the reference count of an xV ever drops to 0, then it will be destroyed and its memory made available for reuse.

This normally doesn't happen at the Perl level unless a variable is undef'ed or the last variable holding a reference to it is changed or overwritten. At the internal level, however, reference counts can be manipulated with the following macros:

```
int SvREFCNT(SV* sv);
SV* SvREFCNT_inc(SV* sv);
void SvREFCNT_dec(SV* sv);
```

However, there is one other function which manipulates the reference count of its argument. The `newRV_inc` function, you will recall, creates a reference to the specified argument. As a side effect, it increments the argument's reference count. If this is not what you want, use `newRV_noinc` instead.

For example, imagine you want to return a reference from an XSUB function. Inside the XSUB routine, you create an SV which initially has a reference count of one. Then you call `newRV_inc`, passing it the just-created SV. This returns the reference as a new SV, but the reference count of the SV you passed to `newRV_inc` has been incremented to two. Now you return the reference from the XSUB routine and forget about the SV. But Perl hasn't! Whenever the returned reference is destroyed, the reference count of the original SV is decreased to one and nothing happens. The SV will hang around without any way to access it until Perl itself terminates. This is a memory leak.

The correct procedure, then, is to use `newRV_noinc` instead of `newRV_inc`. Then, if and when the last reference is destroyed, the reference count of the SV will go to zero and it will be destroyed, stopping any memory leak.

There are some convenience functions available that can help with the destruction of xVs. These functions introduce the concept of "mortality". An xV that is mortal has had its reference count marked to be decremented, but not actually decremented, until "a short time later". Generally the term "short time later" means a single Perl statement, such as a call to an XSUB function. The actual determinant for when mortal xVs have their reference count decremented depends on two macros, SAVETMPS and FREETMPS. See the *perlcall* manpage and the *perlxs* manpage for more details on these macros.

"Mortalization" then is at its simplest a deferred `SvREFCNT_dec`. However, if you mortalize a variable twice, the reference count will later be decremented twice.

You should be careful about creating mortal variables. Strange things can happen if you make the same value mortal within multiple contexts, or if you make a variable mortal multiple times.

To create a mortal variable, use the functions:

```
SV*  sv_newmortal()
SV*  sv_2mortal(SV*)
SV*  sv_mortalcopy(SV*)
```

The first call creates a mortal SV, the second converts an existing SV to a mortal SV (and thus defers a call to `SvREFCNT_dec`), and the third creates a mortal copy of an existing SV.

The mortal routines are not just for SVs -- AVs and HVs can be made mortal by passing their address (type-casted to `SV*`) to the `sv_2mortal` or `sv_mortalcopy` routines.

Stashes and Globs

A "stash" is a hash that contains all of the different objects that are contained within a package. Each key of the stash is a symbol name (shared by all the different types of objects that have the same name), and each value in the hash table is a GV (Glob Value). This GV in turn contains references to the various objects of that name, including (but not limited to) the following:

```
Scalar Value
Array Value
Hash Value
I/O Handle
Format
Subroutine
```

There is a single stash called "PL_defstash" that holds the items that exist in the "main" package. To get at the items in other packages, append the string "::" to the package name. The items in the "Foo" package are in the stash "Foo::" in PL_defstash. The items in the "Bar::Baz" package are in the stash "Baz::" in "Bar::"'s stash.

To get the stash pointer for a particular package, use the function:

```
HV*  gv_stashpv(char* name, I32 create)
HV*  gv_stashsv(SV*, I32 create)
```

The first function takes a literal string, the second uses the string stored in the SV. Remember that a stash is just a hash table, so you get back an `HV*`. The `create` flag will create a new package if it is set.

The name that `gv_stash*v` wants is the name of the package whose symbol table you want. The default package is called `main`. If you have multiply nested packages, pass their names to `gv_stash*v`, separated by `::` as in the Perl language itself.

Alternately, if you have an SV that is a blessed reference, you can find out the stash pointer by using:

```
HV*  SvSTASH(SvRV(SV*));
```

then use the following to get the package name itself:

```
char*  HvNAME(HV* stash);
```

If you need to bless or re-bless an object you can use the following function:

```
SV*  sv_bless(SV*, HV* stash)
```

where the first argument, an `SV*`, must be a reference, and the second argument is a stash. The returned `SV*` can now be used in the same way as any other SV.

For more information on references and blessings, consult the *perlref* manpage.

Double-Typed SVs

Scalar variables normally contain only one type of value, an integer, double, pointer, or reference. Perl will automatically convert the actual scalar data from the stored type into the requested type.

Some scalar variables contain more than one type of scalar data. For example, the variable `$!` contains either the numeric value of `errno` or its string equivalent from either `strerror` or `sys_errlist[]`.

To force multiple data values into an SV, you must do two things: use the `sv_set*v` routines to add the additional scalar type, then set a flag so that Perl will believe it contains more than one type of data. The four macros to set the flags are:

```
SvIOK_on
SvNOK_on
SvPOK_on
SvROK_on
```

The particular macro you must use depends on which `sv_set*v` routine you called first. This is because every `sv_set*v` routine turns on only the bit for the particular type of data being set, and turns off all the rest.

For example, to create a new Perl variable called "dberror" that contains both the numeric and descriptive string error values, you could use the following code:

```
extern int  dberror;
extern char *dberror_list;
```

```
SV* sv = perl_get_sv("dberror", TRUE);
sv_setiv(sv, (IV) dberror);
sv_setpv(sv, dberror_list[dberror]);
SvIOK_on(sv);
```

If the order of `sv_setiv` and `sv_setpv` had been reversed, then the macro `SvPOK_on` would need to be called instead of `SvIOK_on`.

Magic Variables

[This section still under construction. Ignore everything here. Post no bills. Everything not permitted is forbidden.]

Any SV may be magical, that is, it has special features that a normal SV does not have. These features are stored in the SV structure in a linked list of `struct magic`'s, typedef'ed to `MAGIC`.

```
struct magic {
    MAGIC*      mg_moremagic;
    MGVTBL*     mg_virtual;
    U16         mg_private;
    char        mg_type;
    U8          mg_flags;
    SV*         mg_obj;
    char*       mg_ptr;
    I32         mg_len;
};
```

Note this is current as of patchlevel 0, and could change at any time.

Assigning Magic

Perl adds magic to an SV using the sv_magic function:

```
void sv_magic(SV* sv, SV* obj, int how, char* name, I32 namlen);
```

The `sv` argument is a pointer to the SV that is to acquire a new magical feature.

If `sv` is not already magical, Perl uses the `SvUPGRADE` macro to set the `SVt_PVMG` flag for the `sv`. Perl then continues by adding it to the beginning of the linked list of magical features. Any prior entry of the same type of magic is deleted. Note that this can be overridden, and multiple instances of the same type of magic can be associated with an SV.

The `name` and `namlen` arguments are used to associate a string with the magic, typically the name of a variable. `namlen` is stored in the `mg_len` field and if `name` is non-null and `namlen` >= 0 a malloc'd copy of the name is stored in `mg_ptr` field.

The sv_magic function uses `how` to determine which, if any, predefined "Magic Virtual Table" should be assigned to the `mg_virtual` field. See the "Magic Virtual Table" section below. The `how` argument is also stored in the `mg_type` field.

The `obj` argument is stored in the `mg_obj` field of the `MAGIC` structure. If it is not the same as the `sv` argument, the reference count of the `obj` object is incremented. If it is the same, or if the `how` argument is "#", or if it is a NULL pointer, then `obj` is merely stored, without the reference count being incremented.

There is also a function to add magic to an `HV`:

```
void hv_magic(HV *hv, GV *gv, int how);
```

This simply calls `sv_magic` and coerces the `gv` argument into an `SV`.

To remove the magic from an SV, call the function sv_unmagic:

```
void sv_unmagic(SV *sv, int type);
```

The `type` argument should be equal to the `how` value when the `SV` was initially made magical.

Magic Virtual Tables

The `mg_virtual` field in the `MAGIC` structure is a pointer to a `MGVTBL`, which is a structure of function pointers and stands for "Magic Virtual Table" to handle the various operations that might be applied to that variable.

The `MGVTBL` has five pointers to the following routine types:

```
int  (*svt_get)(SV* sv, MAGIC* mg);
int  (*svt_set)(SV* sv, MAGIC* mg);
U32  (*svt_len)(SV* sv, MAGIC* mg);
int  (*svt_clear)(SV* sv, MAGIC* mg);
int  (*svt_free)(SV* sv, MAGIC* mg);
```

This MGVTBL structure is set at compile-time in `perl.h` and there are currently 19 types (or 21 with overloading turned on). These different structures contain pointers to various routines that perform additional actions depending on which function is being called.

```
Function pointer    Action taken
----------------    ------------
svt_get             Do something after the value of the SV is retrieved.
svt_set             Do something after the SV is assigned a value.
svt_len             Report on the SV's length.
svt_clear           Clear something the SV represents.
svt_free            Free any extra storage associated with the SV.
```

For instance, the MGVTBL structure called `vtbl_sv` (which corresponds to an `mg_type` of '\0') contains:

```
{ magic_get, magic_set, magic_len, 0, 0 }
```

Thus, when an SV is determined to be magical and of type '\0', if a get operation is being performed, the routine `magic_get` is called. All the various routines for the various magical types begin with `magic_`.

The current kinds of Magic Virtual Tables are:

```
mg_type  MGVTBL              Type of magic
-------  ------              ----------------------------
\0       vtbl_sv             Special scalar variable
A        vtbl_amagic         %OVERLOAD hash
a        vtbl_amagicelem     %OVERLOAD hash element
c        (none)              Holds overload table (AMT) on stash
B        vtbl_bm             Boyer-Moore (fast string search)
E        vtbl_env            %ENV hash
e        vtbl_envelem        %ENV hash element
f        vtbl_fm             Formline ('compiled' format)
g        vtbl_mglob          m//g target / study()ed string
I        vtbl_isa            @ISA array
i        vtbl_isaelem        @ISA array element
k        vtbl_nkeys          scalar(keys()) lvalue
L        (none)              Debugger %_<filename
l        vtbl_dbline         Debugger %_<filename element
o        vtbl_collxfrm       Locale transformation
P        vtbl_pack           Tied array or hash
p        vtbl_packelem       Tied array or hash element
q        vtbl_packelem       Tied scalar or handle
S        vtbl_sig            %SIG hash
s        vtbl_sigelem        %SIG hash element
t        vtbl_taint          Taintedness
U        vtbl_uvar           Available for use by extensions
v        vtbl_vec            vec() lvalue
x        vtbl_substr         substr() lvalue
y        vtbl_defelem        Shadow "foreach" iterator variable /
                              smart parameter vivification
*        vtbl_glob           GV (typeglob)
#        vtbl_arylen         Array length ($#ary)
.        vtbl_pos            pos() lvalue
~        (none)              Available for use by extensions
```

When an uppercase and lowercase letter both exist in the table, then the uppercase letter is used to represent some kind of composite type (a list or a hash), and the lowercase letter is used to represent an element of that composite type.

The '~' and 'U' magic types are defined specifically for use by extensions and will not be used by perl itself. Extensions can use '~' magic to 'attach' private information to variables (typically objects). This is especially useful because there is no way for normal perl code to corrupt this private information (unlike using extra elements of a hash object).

Similarly, 'U' magic can be used much like *tie()* to call a C function any time a scalar's value is used or changed. The `MAGIC`'s `mg_ptr` field points to a `ufuncs` structure:

```
struct ufuncs {
    I32 (*uf_val)(IV, SV*);
    I32 (*uf_set)(IV, SV*);
    IV uf_index;
};
```

When the SV is read from or written to, the `uf_val` or `uf_set` function will be called with `uf_index` as the first arg and a pointer to the SV as the second. A simple example of how to add 'U' magic is shown below. Note that the ufuncs structure is copied by sv_magic, so you can safely allocate it on the stack.

```
void
Umagic(sv)
    SV *sv;
PREINIT:
    struct ufuncs uf;
CODE:
    uf.uf_val   = &my_get_fn;
    uf.uf_set   = &my_set_fn;
    uf.uf_index = 0;
    sv_magic(sv, 0, 'U', (char*)&uf, sizeof(uf));
```

Note that because multiple extensions may be using '~' or 'U' magic, it is important for extensions to take extra care to avoid conflict. Typically only using the magic on objects blessed into the same class as the extension is sufficient. For '~' magic, it may also be appropriate to add an I32 'signature' at the top of the private data area and check that.

Also note that the `sv_set*()` and `sv_cat*()` functions described earlier do **not** invoke 'set' magic on their targets. This must be done by the user either by calling the `SvSETMAGIC()` macro after calling these functions, or by using one of the `sv_set*_mg()` or `sv_cat*_mg()` functions. Similarly, generic C code must call the `SvGETMAGIC()` macro to invoke any 'get' magic if they use an SV obtained from external sources in functions that don't handle magic. the section on *API LISTING* later in this document identifies such functions. For example, calls to the `sv_cat*()` functions typically need to be followed by `SvSETMAGIC()`, but they don't need a prior `SvGETMAGIC()` since their implementation handles 'get' magic.

Finding Magic

```
MAGIC* mg_find(SV*, int type); /* Finds the magic pointer of that type */
```

This routine returns a pointer to the `MAGIC` structure stored in the SV. If the SV does not have that magical feature, `NULL` is returned. Also, if the SV is not of type SVt_PVMG, Perl may core dump.

```
int mg_copy(SV* sv, SV* nsv, char* key, STRLEN klen);
```

This routine checks to see what types of magic `sv` has. If the mg_type field is an uppercase letter, then the mg_obj is copied to `nsv`, but the mg_type field is changed to be the lowercase letter.

Understanding the Magic of Tied Hashes and Arrays

Tied hashes and arrays are magical beasts of the 'P' magic type.

WARNING: As of the 5.004 release, proper usage of the array and hash access functions requires understanding a few caveats. Some of these caveats are actually considered bugs in the API, to be fixed in later releases, and are bracketed with [MAYCHANGE] below. If you find yourself actually applying such information in this section, be aware that the behavior may change in the future, umm, without warning.

The perl tie function associates a variable with an object that implements the various GET, SET etc methods. To perform the equivalent of the perl tie function from an XSUB, you must mimic this behaviour. The code below carries out the necessary steps − firstly it creates a new hash, and then creates a second hash which it blesses into the class which will implement the tie methods. Lastly it ties the two hashes together, and returns a reference to the new tied hash. Note that the code below does NOT call the TIEHASH method in the MyTie class − see the section on *Calling Perl Routines from within C Programs* for details on how to do this.

```
SV*
mytie()
PREINIT:
    HV *hash;
    HV *stash;
    SV *tie;
CODE:
    hash = newHV();
    tie = newRV_noinc((SV*)newHV());
    stash = gv_stashpv("MyTie", TRUE);
    sv_bless(tie, stash);
    hv_magic(hash, tie, 'P');
    RETVAL = newRV_noinc(hash);
OUTPUT:
    RETVAL
```

The `av_store` function, when given a tied array argument, merely copies the magic of the array onto the value to be "stored", using `mg_copy`. It may also return NULL, indicating that the value did not actually need to be stored in the array. [MAYCHANGE] After a call to `av_store` on a tied array, the caller will usually need to call `mg_set(val)` to actually invoke the perl level "STORE" method on the TIEARRAY object. If `av_store` did return NULL, a call to `SvREFCNT_dec(val)` will also be usually necessary to avoid a memory leak. [/MAYCHANGE]

The previous paragraph is applicable verbatim to tied hash access using the `hv_store` and `hv_store_ent` functions as well.

`av_fetch` and the corresponding hash functions `hv_fetch` and `hv_fetch_ent` actually return an undefined mortal value whose magic has been initialized using `mg_copy`. Note the value so returned does not need to be deallocated, as it is already mortal. [MAYCHANGE] But you will need to call `mg_get()` on the returned value in order to actually invoke the perl level "FETCH" method on the underlying TIE object. Similarly, you may also call `mg_set()` on the return value after possibly assigning a suitable value to it using `sv_setsv`, which will invoke the "STORE" method on the TIE object. [/MAYCHANGE]

[MAYCHANGE] In other words, the array or hash fetch/store functions don't really fetch and store actual values in the case of tied arrays and hashes. They merely call `mg_copy` to attach magic to the values that were meant to be "stored" or "fetched". Later calls to `mg_get` and `mg_set` actually do the job of invoking the TIE methods on the underlying objects. Thus the magic mechanism currently implements a kind of lazy access to arrays and hashes.

Currently (as of perl version 5.004), use of the hash and array access functions requires the user to be aware of whether they are operating on "normal" hashes and arrays, or on their tied variants. The API may be changed to provide more transparent access to both tied and normal data types in future versions. [/MAYCHANGE]

You would do well to understand that the TIEARRAY and TIEHASH interfaces are mere sugar to invoke some perl method calls while using the uniform hash and array syntax. The use of this sugar imposes some overhead (typically about two to four extra opcodes per FETCH/STORE operation, in addition to the creation of all the mortal variables required to invoke the methods). This overhead will be comparatively small if the TIE methods are themselves substantial, but if they are only a few statements long, the overhead will not be insignificant.

Localizing changes

Perl has a very handy construction

```
{
  local $var = 2;
  ...
}
```

This construction is *approximately* equivalent to

```
{
  my $oldvar = $var;
  $var = 2;
  ...
  $var = $oldvar;
}
```

The biggest difference is that the first construction would reinstate the initial value of `$var`, irrespective of how control exits the block: `goto`, `return`, `die`/`eval` etc. It is a little bit more efficient as well.

There is a way to achieve a similar task from C via Perl API: create a *pseudo-block*, and arrange for some changes to be automatically undone at the end of it, either explicit, or via a non-local exit (via *die()*). A *block*−like construct is created by a pair of `ENTER`/`LEAVE` macros (see the section on *Returning a Scalar* in the *perlcall* manpage). Such a construct may be created specially for some important localized task, or an existing one (like boundaries of enclosing Perl subroutine/block, or an existing pair for freeing TMPs) may be used. (In the second case the overhead of additional localization must be almost negligible.) Note that any XSUB is automatically enclosed in an `ENTER`/`LEAVE` pair.

Inside such a *pseudo-block* the following service is available:

`SAVEINT(int i)`

`SAVEIV(IV i)`

`SAVEI32(I32 i)`

`SAVELONG(long i)`

These macros arrange things to restore the value of integer variable `i` at the end of enclosing *pseudo-block*.

`SAVESPTR(s)`

`SAVEPPTR(p)`

These macros arrange things to restore the value of pointers `s` and `p`. `s` must be a pointer of a type which survives conversion to `SV*` and back, `p` should be able to survive conversion to `char*` and back.

`SAVEFREESV(SV *sv)`

The refcount of `sv` would be decremented at the end of *pseudo-block*. This is similar to `sv_2mortal`, which should (?) be used instead.

`SAVEFREEOP(OP *op)`

The `OP *` is *op_free()*ed at the end of *pseudo-block*.

`SAVEFREEPV(p)`

The chunk of memory which is pointed to by `p` is *Safefree()*ed at the end of *pseudo-block*.

`SAVECLEARSV(SV *sv)`

Clears a slot in the current scratchpad which corresponds to `sv` at the end of *pseudo-block*.

`SAVEDELETE(HV *hv, char *key, I32 length)`

The key `key` of `hv` is deleted at the end of *pseudo-block*. The string pointed to by `key` is *Safefree()*ed. If one has a *key* in short-lived storage, the corresponding string may be reallocated like this:

```
SAVEDELETE(PL_defstash, savepv(tmpbuf), strlen(tmpbuf));
```

`SAVEDESTRUCTOR(f,p)`
: At the end of *pseudo-block* the function `f` is called with the only argument (of type `void*`) `p`.

`SAVESTACK_POS()`
: The current offset on the Perl internal stack (cf. `SP`) is restored at the end of *pseudo-block*.

The following API list contains functions, thus one needs to provide pointers to the modifiable data explicitly (either C pointers, or Perlish `GV *`s). Where the above macros take `int`, a similar function takes `int *`.

`SV* save_scalar(GV *gv)`
: Equivalent to Perl code `local $gv`.

`AV* save_ary(GV *gv)`

`HV* save_hash(GV *gv)`
: Similar to `save_scalar`, but localize `@gv` and `%gv`.

`void save_item(SV *item)`
: Duplicates the current value of `SV`, on the exit from the current `ENTER`/`LEAVE` *pseudo-block* will restore the value of `SV` using the stored value.

`void save_list(SV **sarg, I32 maxsarg)`
: A variant of `save_item` which takes multiple arguments via an array `sarg` of `SV*` of length `maxsarg`.

`SV* save_svref(SV **sptr)`
: Similar to `save_scalar`, but will reinstate a `SV *`.

`void save_aptr(AV **aptr)`

`void save_hptr(HV **hptr)`
: Similar to `save_svref`, but localize `AV *` and `HV *`.

The `Alias` module implements localization of the basic types within the *caller's scope*. People who are interested in how to localize things in the containing scope should take a look there too.

Subroutines

XSUBs and the Argument Stack

The XSUB mechanism is a simple way for Perl programs to access C subroutines. An XSUB routine will have a stack that contains the arguments from the Perl program, and a way to map from the Perl data structures to a C equivalent.

The stack arguments are accessible through the `ST(n)` macro, which returns the `n`'th stack argument. Argument 0 is the first argument passed in the Perl subroutine call. These arguments are `SV*`, and can be used anywhere an `SV*` is used.

Most of the time, output from the C routine can be handled through use of the RETVAL and OUTPUT directives. However, there are some cases where the argument stack is not already long enough to handle all the return values. An example is the POSIX *tzname()* call, which takes no arguments, but returns two, the local time zone's standard and summer time abbreviations.

To handle this situation, the PPCODE directive is used and the stack is extended using the macro:

```
EXTEND(SP, num);
```

where `SP` is the macro that represents the local copy of the stack pointer, and `num` is the number of elements the stack should be extended by.

Now that there is room on the stack, values can be pushed on it using the macros to push IVs, doubles, strings, and SV pointers respectively:

```
PUSHi(IV)
PUSHn(double)
PUSHp(char*, I32)
PUSHs(SV*)
```

And now the Perl program calling `tzname`, the two values will be assigned as in:

```
($standard_abbrev, $summer_abbrev) = POSIX::tzname;
```

An alternate (and possibly simpler) method to pushing values on the stack is to use the macros:

```
XPUSHi(IV)
XPUSHn(double)
XPUSHp(char*, I32)
XPUSHs(SV*)
```

These macros automatically adjust the stack for you, if needed. Thus, you do not need to call `EXTEND` to extend the stack.

For more information, consult the *perlxs* manpage and the *perlxstut* manpage.

Calling Perl Routines from within C Programs

There are four routines that can be used to call a Perl subroutine from within a C program. These four are:

```
I32  perl_call_sv(SV*, I32);
I32  perl_call_pv(char*, I32);
I32  perl_call_method(char*, I32);
I32  perl_call_argv(char*, I32, register char**);
```

The routine most often used is `perl_call_sv`. The `SV*` argument contains either the name of the Perl subroutine to be called, or a reference to the subroutine. The second argument consists of flags that control the context in which the subroutine is called, whether or not the subroutine is being passed arguments, how errors should be trapped, and how to treat return values.

All four routines return the number of arguments that the subroutine returned on the Perl stack.

When using any of these routines (except `perl_call_argv`), the programmer must manipulate the Perl stack. These include the following macros and functions:

```
dSP
SP
PUSHMARK()
PUTBACK
SPAGAIN
ENTER
SAVETMPS
FREETMPS
LEAVE
XPUSH*()
POP*()
```

For a detailed description of calling conventions from C to Perl, consult the *perlcall* manpage.

Memory Allocation

All memory meant to be used with the Perl API functions should be manipulated using the macros described in this section. The macros provide the necessary transparency between differences in the actual malloc implementation that is used within perl.

It is suggested that you enable the version of malloc that is distributed with Perl. It keeps pools of various sizes of unallocated memory in order to satisfy allocation requests more quickly. However, on some platforms, it may cause spurious malloc or free errors.

```
New(x, pointer, number, type);
Newc(x, pointer, number, type, cast);
Newz(x, pointer, number, type);
```

These three macros are used to initially allocate memory.

The first argument `x` was a "magic cookie" that was used to keep track of who called the macro, to help when debugging memory problems. However, the current code makes no use of this feature (most Perl developers now use run-time memory checkers), so this argument can be any number.

The second argument `pointer` should be the name of a variable that will point to the newly allocated memory.

The third and fourth arguments `number` and `type` specify how many of the specified type of data structure should be allocated. The argument `type` is passed to `sizeof`. The final argument to `Newc`, `cast`, should be used if the `pointer` argument is different from the `type` argument.

Unlike the `New` and `Newc` macros, the `Newz` macro calls `memzero` to zero out all the newly allocated memory.

```
Renew(pointer, number, type);
Renewc(pointer, number, type, cast);
Safefree(pointer)
```

These three macros are used to change a memory buffer size or to free a piece of memory no longer needed. The arguments to `Renew` and `Renewc` match those of `New` and `Newc` with the exception of not needing the "magic cookie" argument.

```
Move(source, dest, number, type);
Copy(source, dest, number, type);
Zero(dest, number, type);
```

These three macros are used to move, copy, or zero out previously allocated memory. The `source` and `dest` arguments point to the source and destination starting points. Perl will move, copy, or zero out `number` instances of the size of the `type` data structure (using the `sizeof` function).

PerlIO

The most recent development releases of Perl has been experimenting with removing Perl's dependency on the "normal" standard I/O suite and allowing other stdio implementations to be used. This involves creating a new abstraction layer that then calls whichever implementation of stdio Perl was compiled with. All XSUBs should now use the functions in the PerlIO abstraction layer and not make any assumptions about what kind of stdio is being used.

For a complete description of the PerlIO abstraction, consult the *perlapio* manpage.

Putting a C value on Perl stack

A lot of opcodes (this is an elementary operation in the internal perl stack machine) put an SV* on the stack. However, as an optimization the corresponding SV is (usually) not recreated each time. The opcodes reuse specially assigned SVs (*target*s) which are (as a corollary) not constantly freed/created.

Each of the targets is created only once (but see the section on *Scratchpads and recursion* below), and when an opcode needs to put an integer, a double, or a string on stack, it just sets the corresponding parts of its *target* and puts the *target* on stack.

The macro to put this target on stack is `PUSHTARG`, and it is directly used in some opcodes, as well as indirectly in zillions of others, which use it via `(X)PUSH[pni]`.

Scratchpads

The question remains on when the SVs which are *target*s for opcodes are created. The answer is that they are created when the current unit -- a subroutine or a file (for opcodes for statements outside of subroutines) -- is compiled. During this time a special anonymous Perl array is created, which is called a scratchpad for the current unit.

A scratchpad keeps SVs which are lexicals for the current unit and are targets for opcodes. One can deduce that an SV lives on a scratchpad by looking on its flags: lexicals have `SVs_PADMY` set, and *target*s have `SVs_PADTMP` set.

The correspondence between OPs and *target*s is not 1-to-1. Different OPs in the compile tree of the unit can use the same target, if this would not conflict with the expected life of the temporary.

Scratchpads and recursion

In fact it is not 100% true that a compiled unit contains a pointer to the scratchpad AV. In fact it contains a pointer to an AV of (initially) one element, and this element is the scratchpad AV. Why do we need an extra level of indirection?

The answer is **recursion**, and maybe (sometime soon) **threads**. Both these can create several execution pointers going into the same subroutine. For the subroutine-child not write over the temporaries for the subroutine-parent (lifespan of which covers the call to the child), the parent and the child should have different scratchpads. (*And* the lexicals should be separate anyway!)

So each subroutine is born with an array of scratchpads (of length 1). On each entry to the subroutine it is checked that the current depth of the recursion is not more than the length of this array, and if it is, new scratchpad is created and pushed into the array.

The *target*s on this scratchpad are `undef`s, but they are already marked with correct flags.

Compiled code

Code tree

Here we describe the internal form your code is converted to by Perl. Start with a simple example:

```
$a = $b + $c;
```

This is converted to a tree similar to this one:

```
             assign-to
           /           \
          +             $a
        /   \
      $b     $c
```

(but slightly more complicated). This tree reflects the way Perl parsed your code, but has nothing to do with the execution order. There is an additional ‘‘thread’’ going through the nodes of the tree which shows the order of execution of the nodes. In our simplified example above it looks like:

```
$b ---> $c ---> + ---> $a ---> assign-to
```

But with the actual compile tree for `$a = $b + $c` it is different: some nodes *optimized away*. As a corollary, though the actual tree contains more nodes than our simplified example, the execution order is the same as in our example.

Examining the tree

If you have your perl compiled for debugging (usually done with `-D optimize=-g` on `Configure` command line), you may examine the compiled tree by specifying `-Dx` on the Perl command line. The output takes several lines per node, and for `$b+$c` it looks like this:

```
5           TYPE = add  ===> 6
            TARG = 1
            FLAGS = (SCALAR,KIDS)
            {
                TYPE = null  ===> (4)
                  (was rv2sv)
                FLAGS = (SCALAR,KIDS)
                {
3                   TYPE = gvsv  ===> 4
                    FLAGS = (SCALAR)
                    GV = main::b
                }
            }
            {
                TYPE = null  ===> (5)
                  (was rv2sv)
                FLAGS = (SCALAR,KIDS)
                {
4                   TYPE = gvsv  ===> 5
                    FLAGS = (SCALAR)
                    GV = main::c
                }
            }
```

This tree has 5 nodes (one per `TYPE` specifier), only 3 of them are not optimized away (one per number in the left column). The immediate children of the given node correspond to {} pairs on the same level of indentation, thus this listing corresponds to the tree:

```
     add
    /   \
 null   null
  |       |
 gvsv   gvsv
```

The execution order is indicated by `===>` marks, thus it is `3 4 5 6` (node `6` is not included into above listing), i.e., `gvsv gvsv add whatever`.

Compile pass 1: check routines

The tree is created by the *pseudo-compiler* while yacc code feeds it the constructions it recognizes. Since yacc works bottom-up, so does the first pass of perl compilation.

What makes this pass interesting for perl developers is that some optimization may be performed on this pass. This is optimization by so-called *check routines*. The correspondence between node names and corresponding check routines is described in *opcode.pl* (do not forget to run `make regen_headers` if you modify this file).

A check routine is called when the node is fully constructed except for the execution-order thread. Since at this time there are no back-links to the currently constructed node, one can do most any operation to the top-level node, including freeing it and/or creating new nodes above/below it.

The check routine returns the node which should be inserted into the tree (if the top-level node was not modified, check routine returns its argument).

By convention, check routines have names `ck_*`. They are usually called from `new*OP` subroutines (or `convert`) (which in turn are called from *perly.y*).

Compile pass 1a: constant folding

Immediately after the check routine is called the returned node is checked for being compile-time executable. If it is (the value is judged to be constant) it is immediately executed, and a *constant* node with the "return value" of the corresponding subtree is substituted instead. The subtree is deleted.

If constant folding was not performed, the execution-order thread is created.

Compile pass 2: context propagation

When a context for a part of compile tree is known, it is propagated down through the tree. At this time the context can have 5 values (instead of 2 for runtime context): void, boolean, scalar, list, and lvalue. In contrast with the pass 1 this pass is processed from top to bottom: a node's context determines the context for its children.

Additional context-dependent optimizations are performed at this time. Since at this moment the compile tree contains back-references (via "thread" pointers), nodes cannot be *free()*d now. To allow optimized-away nodes at this stage, such nodes are *null()*ified instead of *free()*ing (i.e. their type is changed to OP_NULL).

Compile pass 3: peephole optimization

After the compile tree for a subroutine (or for an `eval` or a file) is created, an additional pass over the code is performed. This pass is neither top-down or bottom-up, but in the execution order (with additional complications for conditionals). These optimizations are done in the subroutine *peep()*. Optimizations performed at this stage are subject to the same restrictions as in the pass 2.

API LISTING

This is a listing of functions, macros, flags, and variables that may be useful to extension writers or that may be found while reading other extensions.

Note that all Perl API global variables must be referenced with the `PL_` prefix. Some macros are provided for compatibility with the older, unadorned names, but this support will be removed in a future release.

It is strongly recommended that all Perl API functions that don't begin with `perl` be referenced with an explicit `Perl_` prefix.

The sort order of the listing is case insensitive, with any occurrences of '_' ignored for the purpose of sorting.

av_clear Clears an array, making it empty. Does not free the memory used by the array itself.

```
void    av_clear (AV* ar)
```

av_extend
Pre-extend an array. The `key` is the index to which the array should be extended.

```
void    av_extend (AV* ar, I32 key)
```

av_fetch Returns the SV at the specified index in the array. The `key` is the index. If `lval` is set then the fetch will be part of a store. Check that the return value is non-null before dereferencing it to a `SV*`.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied arrays.

```
SV**    av_fetch (AV* ar, I32 key, I32 lval)
```

AvFILL Same as `av_len()`. Deprecated, use `av_len()` instead.

av_len
: Returns the highest index in the array. Returns −1 if the array is empty.

```
I32     av_len (AV* ar)
```

av_make
: Creates a new AV and populates it with a list of SVs. The SVs are copied into the array, so they may be freed after the call to av_make. The new AV will have a reference count of 1.

```
AV*     av_make (I32 size, SV** svp)
```

av_pop
: Pops an SV off the end of the array. Returns `&PL_sv_undef` if the array is empty.

```
SV*     av_pop (AV* ar)
```

av_push
: Pushes an SV onto the end of the array. The array will grow automatically to accommodate the addition.

```
void    av_push (AV* ar, SV* val)
```

av_shift
: Shifts an SV off the beginning of the array.

```
SV*     av_shift (AV* ar)
```

av_store
: Stores an SV in an array. The array index is specified as `key`. The return value will be NULL if the operation failed or if the value did not need to be actually stored within the array (as in the case of tied arrays). Otherwise it can be dereferenced to get the original `SV*`. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied arrays.

```
SV**    av_store (AV* ar, I32 key, SV* val)
```

av_undef
: Undefines the array. Frees the memory used by the array itself.

```
void    av_undef (AV* ar)
```

av_unshift
: Unshift the given number of `undef` values onto the beginning of the array. The array will grow automatically to accommodate the addition. You must then use `av_store` to assign values to these new elements.

```
void    av_unshift (AV* ar, I32 num)
```

CLASS
: Variable which is setup by `xsubpp` to indicate the class name for a C++ XS constructor. This is always a `char*`. See `THIS` and the section on *Using XS With C++* in the *perlxs* manpage.

Copy
: The XSUB−writer's interface to the C `memcpy` function. The `s` is the source, `d` is the destination, `n` is the number of items, and `t` is the type. May fail on overlapping copies. See also `Move`.

```
void    Copy( s, d, n, t )
```

croak This is the XSUB−writer's interface to Perl's `die` function. Use this function the same way you use the C `printf` function. See `warn`.

CvSTASH
Returns the stash of the CV.

```
HV*     CvSTASH( SV* sv )
```

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`.

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`. The sub name can be found by

```
SvPV( GvSV( PL_DBsub ), len )
```

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`.

dMARK Declare a stack marker variable, `mark`, for the XSUB. See `MARK` and `dORIGMARK`.

dORIGMARK
Saves the original stack mark for the XSUB. See `ORIGMARK`.

PL_dowarn
The C variable which corresponds to Perl's $^W warning variable.

dSP Declares a local copy of perl's stack pointer for the XSUB, available via the `SP` macro. See `SP`.

dXSARGS
Sets up stack and mark pointers for an XSUB, calling dSP and dMARK. This is usually handled automatically by `xsubpp`. Declares the `items` variable to indicate the number of items on the stack.

dXSI32 Sets up the `ix` variable for an XSUB which has aliases. This is usually handled automatically by `xsubpp`.

do_binmode
Switches filehandle to binmode. `iotype` is what `IoTYPE(io)` would contain.

```
do_binmode(fp, iotype, TRUE);
```

ENTER Opening bracket on a callback. See `LEAVE` and the *perlcall* manpage.

```
ENTER;
```

EXTEND
Used to extend the argument stack for an XSUB's return values.

```
EXTEND( sp, int x )
```

fbm_compile
Analyses the string in order to make fast searches on it using *fbm_instr()* -- the Boyer-Moore algorithm.

```
void    fbm_compile(SV* sv, U32 flags)
```

fbm_instr
Returns the location of the SV in the string delimited by `str` and `strend`. It returns `Nullch` if the string can't be found. The `sv` does not have to be fbm_compiled, but the search will not be as fast then.

```
char*   fbm_instr(char *str, char *strend, SV *sv, U32 flags)
```

FREETMPS
Closing bracket for temporaries on a callback. See `SAVETMPS` and the *perlcall* manpage.

```
FREETMPS;
```

G_ARRAY
Used to indicate array context. See `GIMME_V`, `GIMME` and the *perlcall* manpage.

G_DISCARD
Indicates that arguments returned from a callback should be discarded. See the *perlcall* manpage.

G_EVAL
Used to force a Perl `eval` wrapper around a callback. See the *perlcall* manpage.

GIMME
A backward-compatible version of `GIMME_V` which can only return `G_SCALAR` or `G_ARRAY`; in a void context, it returns `G_SCALAR`.

GIMME_V
The XSUB−writer's equivalent to Perl's `wantarray`. Returns `G_VOID`, `G_SCALAR` or `G_ARRAY` for void, scalar or array context, respectively.

G_NOARGS
Indicates that no arguments are being sent to a callback. See the *perlcall* manpage.

G_SCALAR
Used to indicate scalar context. See `GIMME_V`, `GIMME`, and the *perlcall* manpage.

gv_fetchmeth
Returns the glob with the given `name` and a defined subroutine or `NULL`. The glob lives in the given `stash`, or in the stashes accessible via `@ISA` and `@UNIVERSAL`.

The argument `level` should be either 0 or −1. If `level==0`, as a side-effect creates a glob with the given `name` in the given `stash` which in the case of success contains an alias for the subroutine, and sets up caching info for this glob. Similarly for all the searched stashes.

This function grants `"SUPER"` token as a postfix of the stash name.

The GV returned from `gv_fetchmeth` may be a method cache entry, which is not visible to Perl code. So when calling `perl_call_sv`, you should not use the GV directly; instead, you should use the method's CV, which can be obtained from the GV with the `GvCV` macro.

```
GV*     gv_fetchmeth (HV* stash, char* name, STRLEN len, I32 level)
```

gv_fetchmethod

gv_fetchmethod_autoload
Returns the glob which contains the subroutine to call to invoke the method on the `stash`. In fact in the presence of autoloading this may be the glob for "AUTOLOAD". In this case the corresponding variable `$AUTOLOAD` is already setup.

The third parameter of `gv_fetchmethod_autoload` determines whether AUTOLOAD lookup is performed if the given method is not present: non-zero means yes, look for

AUTOLOAD; zero means no, don't look for AUTOLOAD. Calling `gv_fetchmethod` is equivalent to calling `gv_fetchmethod_autoload` with a non-zero `autoload` parameter.

These functions grant `"SUPER"` token as a prefix of the method name.

Note that if you want to keep the returned glob for a long time, you need to check for it being "AUTOLOAD", since at the later time the call may load a different subroutine due to `$AUTOLOAD` changing its value. Use the glob created via a side effect to do this.

These functions have the same side-effects and as `gv_fetchmeth` with `level==0`. `name` should be writable if contains `':'` or `'\''`. The warning against passing the GV returned by `gv_fetchmeth` to `perl_call_sv` apply equally to these functions.

```
GV*     gv_fetchmethod (HV* stash, char* name)
GV*     gv_fetchmethod_autoload (HV* stash, char* name, I32 autoload)
```

G_VOID
: Used to indicate void context. See `GIMME_V` and the *perlcall* manpage.

gv_stashpv
: Returns a pointer to the stash for a specified package. If `create` is set then the package will be created if it does not already exist. If `create` is not set and the package does not exist then NULL is returned.

```
HV*     gv_stashpv (char* name, I32 create)
```

gv_stashsv
: Returns a pointer to the stash for a specified package. See `gv_stashpv`.

```
HV*     gv_stashsv (SV* sv, I32 create)
```

GvSV
: Return the SV from the GV.

HEf_SVKEY
: This flag, used in the length slot of hash entries and magic structures, specifies the structure contains a `SV*` pointer where a `char*` pointer is to be expected. (For information only—not to be used).

HeHASH
: Returns the computed hash stored in the hash entry.

```
U32     HeHASH(HE* he)
```

HeKEY
: Returns the actual pointer stored in the key slot of the hash entry. The pointer may be either `char*` or `SV*`, depending on the value of `HeKLEN()`. Can be assigned to. The `HePV()` or `HeSVKEY()` macros are usually preferable for finding the value of a key.

```
char*   HeKEY(HE* he)
```

HeKLEN
: If this is negative, and amounts to `HEf_SVKEY`, it indicates the entry holds an `SV*` key. Otherwise, holds the actual length of the key. Can be assigned to. The `HePV()` macro is usually preferable for finding key lengths.

```
int     HeKLEN(HE* he)
```

HePV Returns the key slot of the hash entry as a `char*` value, doing any necessary dereferencing of possibly `SV*` keys. The length of the string is placed in `len` (this is a macro, so do *not* use `&len`). If you do not care about what the length of the key is, you may use the global variable `PL_na`, though this is rather less efficient than using a local variable. Remember though, that hash keys in perl are free to contain embedded nulls, so using `strlen()` or similar is not a good way to find the length of hash keys. This is very similar to the `SvPV()` macro described elsewhere in this document.

```
char*   HePV(HE* he, STRLEN len)
```

HeSVKEY
Returns the key as an `SV*`, or `Nullsv` if the hash entry does not contain an `SV*` key.

```
HeSVKEY(HE* he)
```

HeSVKEY_force
Returns the key as an `SV*`. Will create and return a temporary mortal `SV*` if the hash entry contains only a `char*` key.

```
HeSVKEY_force(HE* he)
```

HeSVKEY_set
Sets the key to a given `SV*`, taking care to set the appropriate flags to indicate the presence of an `SV*` key, and returns the same `SV*`.

```
HeSVKEY_set(HE* he, SV* sv)
```

HeVAL Returns the value slot (type `SV*`) stored in the hash entry.

```
HeVAL(HE* he)
```

hv_clear Clears a hash, making it empty.

```
void    hv_clear (HV* tb)
```

hv_delete
Deletes a key/value pair in the hash. The value SV is removed from the hash and returned to the caller. The `klen` is the length of the key. The `flags` value will normally be zero; if set to G_DISCARD then NULL will be returned.

```
SV*     hv_delete (HV* tb, char* key, U32 klen, I32 flags)
```

hv_delete_ent
Deletes a key/value pair in the hash. The value SV is removed from the hash and returned to the caller. The `flags` value will normally be zero; if set to G_DISCARD then NULL will be returned. `hash` can be a valid precomputed hash value, or 0 to ask for it to be computed.

```
SV*     hv_delete_ent (HV* tb, SV* key, I32 flags, U32 hash)
```

hv_exists
Returns a boolean indicating whether the specified hash key exists. The `klen` is the length of the key.

```
bool    hv_exists (HV* tb, char* key, U32 klen)
```

hv_exists_ent
Returns a boolean indicating whether the specified hash key exists. `hash` can be a valid precomputed hash value, or 0 to ask for it to be computed.

```
bool    hv_exists_ent (HV* tb, SV* key, U32 hash)
```

hv_fetch
Returns the SV which corresponds to the specified key in the hash. The `klen` is the length of the key. If `lval` is set then the fetch will be part of a store. Check that the return value is non-null before dereferencing it to a `SV*`.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied hashes.

```
SV**    hv_fetch (HV* tb, char* key, U32 klen, I32 lval)
```

hv_fetch_ent
Returns the hash entry which corresponds to the specified key in the hash. `hash` must be a valid precomputed hash number for the given `key`, or 0 if you want the function to compute it. IF `lval` is set then the fetch will be part of a store. Make sure the return value is non-null before accessing it. The return value when `tb` is a tied hash is a pointer to a static location, so be sure to make a copy of the structure if you need to store it somewhere.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied hashes.

```
HE*     hv_fetch_ent  (HV* tb, SV* key, I32 lval, U32 hash)
```

hv_iterinit
Prepares a starting point to traverse a hash table.

```
I32     hv_iterinit (HV* tb)
```

Returns the number of keys in the hash (i.e. the same as `HvKEYS(tb)`). The return value is currently only meaningful for hashes without tie magic.

NOTE: Before version 5.004_65, `hv_iterinit` used to return the number of hash buckets that happen to be in use. If you still need that esoteric value, you can get it through the macro `HvFILL(tb)`.

hv_iterkey
Returns the key from the current position of the hash iterator. See `hv_iterinit`.

```
char*   hv_iterkey (HE* entry, I32* retlen)
```

hv_iterkeysv
Returns the key as an `SV*` from the current position of the hash iterator. The return value will always be a mortal copy of the key. Also see `hv_iterinit`.

```
SV*     hv_iterkeysv  (HE* entry)
```

hv_iternext
Returns entries from a hash iterator. See `hv_iterinit`.

```
HE*     hv_iternext (HV* tb)
```

hv_iternextsv

Performs an `hv_iternext`, `hv_iterkey`, and `hv_iterval` in one operation.

```
SV*     hv_iternextsv (HV* hv, char** key, I32* retlen)
```

hv_iterval

Returns the value from the current position of the hash iterator. See `hv_iterkey`.

```
SV*     hv_iterval (HV* tb, HE* entry)
```

hv_magic

Adds magic to a hash. See `sv_magic`.

```
void    hv_magic (HV* hv, GV* gv, int how)
```

HvNAME

Returns the package name of a stash. See `SvSTASH`, `CvSTASH`.

```
char*   HvNAME (HV* stash)
```

hv_store

Stores an SV in a hash. The hash key is specified as `key` and `klen` is the length of the key. The `hash` parameter is the precomputed hash value; if it is zero then Perl will compute it. The return value will be NULL if the operation failed or if the value did not need to be actually stored within the hash (as in the case of tied hashes). Otherwise it can be dereferenced to get the original `SV*`. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied hashes.

```
SV**    hv_store (HV* tb, char* key, U32 klen, SV* val, U32 hash)
```

hv_store_ent

Stores `val` in a hash. The hash key is specified as `key`. The `hash` parameter is the precomputed hash value; if it is zero then Perl will compute it. The return value is the new hash entry so created. It will be NULL if the operation failed or if the value did not need to be actually stored within the hash (as in the case of tied hashes). Otherwise the contents of the return value can be accessed using the `He???` macros described here. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL.

See the section on *Understanding the Magic of Tied Hashes and Arrays* for more information on how to use this function on tied hashes.

```
HE*     hv_store_ent  (HV* tb, SV* key, SV* val, U32 hash)
```

hv_undef

Undefines the hash.

```
void    hv_undef (HV* tb)
```

isALNUM

Returns a boolean indicating whether the C `char` is an ascii alphanumeric character or digit.

```
int     isALNUM (char c)
```

isALPHA
Returns a boolean indicating whether the C `char` is an ascii alphabetic character.

```
int     isALPHA (char c)
```

isDIGIT Returns a boolean indicating whether the C `char` is an ascii digit.

```
int     isDIGIT (char c)
```

isLOWER
Returns a boolean indicating whether the C `char` is a lowercase character.

```
int     isLOWER (char c)
```

isSPACE
Returns a boolean indicating whether the C `char` is whitespace.

```
int     isSPACE (char c)
```

isUPPER
Returns a boolean indicating whether the C `char` is an uppercase character.

```
int     isUPPER (char c)
```

items Variable which is setup by `xsubpp` to indicate the number of items on the stack. See the section on *Variable-length Parameter Lists* in the *perlxs* manpage.

ix Variable which is setup by `xsubpp` to indicate which of an XSUB's aliases was used to invoke it. See the section on *The ALIAS: Keyword* in the *perlxs* manpage.

LEAVE Closing bracket on a callback. See `ENTER` and the *perlcall* manpage.

```
LEAVE;
```

looks_like_number
Test if an the content of an SV looks like a number (or is a number).

```
int     looks_like_number(SV*)
```

MARK Stack marker variable for the XSUB. See `dMARK`.

mg_clear
Clear something magical that the SV represents. See `sv_magic`.

```
int     mg_clear (SV* sv)
```

mg_copy
Copies the magic from one SV to another. See `sv_magic`.

```
int     mg_copy (SV *, SV *, char *, STRLEN)
```

mg_find Finds the magic pointer for type matching the SV. See `sv_magic`.

```
MAGIC*  mg_find (SV* sv, int type)
```

mg_free Free any magic storage used by the SV. See `sv_magic`.

```
int     mg_free (SV* sv)
```

mg_get Do magic after a value is retrieved from the SV. See `sv_magic`.

```
int     mg_get (SV* sv)
```

mg_len Report on the SV's length. See `sv_magic`.

```
U32     mg_len (SV* sv)
```

mg_magical
Turns on the magical status of an SV. See `sv_magic`.

```
void    mg_magical (SV* sv)
```

mg_set Do magic after a value is assigned to the SV. See `sv_magic`.

```
int     mg_set (SV* sv)
```

modglobal
`modglobal` is a general purpose, interpreter global HV for use by extensions that need to keep information on a per-interpreter basis. In a pinch, it can also be used as a symbol table for extensions to share data among each other. It is a good idea to use keys prefixed by the package name of the extension that owns the data.

Move The XSUB−writer's interface to the C `memmove` function. The `s` is the source, `d` is the destination, `n` is the number of items, and `t` is the type. Can do overlapping moves. See also `Copy`.

```
void    Move( s, d, n, t )
```

PL_na A convenience variable which is typically used with `SvPV` when one doesn't care about the length of the string. It is usually more efficient to declare a local variable and use that instead.

New The XSUB−writer's interface to the C `malloc` function.

```
void*   New( x, void *ptr, int size, type )
```

newAV Creates a new AV. The reference count is set to 1.

```
AV*     newAV (void)
```

Newc The XSUB−writer's interface to the C `malloc` function, with cast.

```
void*   Newc( x, void *ptr, int size, type, cast )
```

newCONSTSUB
Creates a constant sub equivalent to Perl `sub FOO () { 123 }` which is eligible for inlining at compile-time.

```
void    newCONSTSUB(HV* stash, char* name, SV* sv)
```

newHV Creates a new HV. The reference count is set to 1.

```
HV*     newHV (void)
```

newRV_inc

Creates an RV wrapper for an SV. The reference count for the original SV is incremented.

```
SV*     newRV_inc (SV* ref)
```

For historical reasons, ‘‘newRV’’ is a synonym for ‘‘newRV_inc’’.

newRV_noinc

Creates an RV wrapper for an SV. The reference count for the original SV is **not** incremented.

```
SV*     newRV_noinc (SV* ref)
```

NEWSV

Creates a new SV. A non-zero `len` parameter indicates the number of bytes of preallocated string space the SV should have. An extra byte for a tailing NUL is also reserved. (SvPOK is not set for the SV even if string space is allocated.) The reference count for the new SV is set to 1. `id` is an integer id between 0 and 1299 (used to identify leaks).

```
SV*     NEWSV (int id, STRLEN len)
```

newSViv

Creates a new SV and copies an integer into it. The reference count for the SV is set to 1.

```
SV*     newSViv (IV i)
```

newSVnv

Creates a new SV and copies a double into it. The reference count for the SV is set to 1.

```
SV*     newSVnv (NV i)
```

newSVpv

Creates a new SV and copies a string into it. The reference count for the SV is set to 1. If `len` is zero then Perl will compute the length.

```
SV*     newSVpv (char* s, STRLEN len)
```

newSVpvf

Creates a new SV an initialize it with the string formatted like `sprintf`.

```
SV*     newSVpvf(const char* pat, ...);
```

newSVpvn

Creates a new SV and copies a string into it. The reference count for the SV is set to 1. If `len` is zero then Perl will create a zero length string.

```
SV*     newSVpvn (char* s, STRLEN len)
```

newSVrv

Creates a new SV for the RV, `rv`, to point to. If `rv` is not an RV then it will be upgraded to one. If `classname` is non-null then the new SV will be blessed in the specified package. The new SV is returned and its reference count is 1.

```
SV*     newSVrv (SV* rv, char* classname)
```

newSVsv

Creates a new SV which is an exact duplicate of the original SV.

```
SV*     newSVsv (SV* old)
```

newXS Used by `xsubpp` to hook up XSUBs as Perl subs.

newXSproto
Used by `xsubpp` to hook up XSUBs as Perl subs. Adds Perl prototypes to the subs.

Newz The XSUB−writer's interface to the C `malloc` function. The allocated memory is zeroed with `memzero`.

```
void*   Newz( x, void *ptr, int size, type )
```

Nullav Null AV pointer.

Nullch Null character pointer.

Nullcv Null CV pointer.

Nullhv Null HV pointer.

Nullsv Null SV pointer.

ORIGMARK
The original stack mark for the XSUB. See `dORIGMARK`.

perl_alloc
Allocates a new Perl interpreter. See the *perlembed* manpage.

perl_call_argv
Performs a callback to the specified Perl sub. See the *perlcall* manpage.

```
I32     perl_call_argv (char* subname, I32 flags, char** argv)
```

perl_call_method
Performs a callback to the specified Perl method. The blessed object must be on the stack. See the *perlcall* manpage.

```
I32     perl_call_method (char* methname, I32 flags)
```

perl_call_pv
Performs a callback to the specified Perl sub. See the *perlcall* manpage.

```
I32     perl_call_pv (char* subname, I32 flags)
```

perl_call_sv
Performs a callback to the Perl sub whose name is in the SV. See the *perlcall* manpage.

```
I32     perl_call_sv (SV* sv, I32 flags)
```

perl_construct
Initializes a new Perl interpreter. See the *perlembed* manpage.

perl_destruct
Shuts down a Perl interpreter. See the *perlembed* manpage.

perl_eval_sv
Tells Perl to `eval` the string in the SV.

```
I32     perl_eval_sv (SV* sv, I32 flags)
```

perl_eval_pv
Tells Perl to `eval` the given string and return an SV* result.

```
SV*     perl_eval_pv (char* p, I32 croak_on_error)
```

perl_free
Releases a Perl interpreter. See the *perlembed* manpage.

perl_get_av
Returns the AV of the specified Perl array. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

```
AV*     perl_get_av (char* name, I32 create)
```

perl_get_cv
Returns the CV of the specified Perl sub. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

```
CV*     perl_get_cv (char* name, I32 create)
```

perl_get_hv
Returns the HV of the specified Perl hash. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

```
HV*     perl_get_hv (char* name, I32 create)
```

perl_get_sv
Returns the SV of the specified Perl scalar. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

```
SV*     perl_get_sv (char* name, I32 create)
```

perl_parse
Tells a Perl interpreter to parse a Perl script. See the *perlembed* manpage.

perl_require_pv
Tells Perl to `require` a module.

```
void    perl_require_pv (char* pv)
```

perl_run Tells a Perl interpreter to run. See the *perlembed* manpage.

POPi Pops an integer off the stack.

```
int     POPi()
```

POPl Pops a long off the stack.

```
long    POPl()
```

POPp Pops a string off the stack.

```
char*   POPp()
```

POPn Pops a double off the stack.

```
double  POPn()
```

POPs Pops an SV off the stack.

```
SV*     POPs()
```

PUSHMARK
Opening bracket for arguments on a callback. See `PUTBACK` and the *perlcall* manpage.

```
PUSHMARK(p)
```

PUSHi Push an integer onto the stack. The stack must have room for this element. Handles 'set' magic. See `XPUSHi`.

```
void    PUSHi(int d)
```

PUSHn Push a double onto the stack. The stack must have room for this element. Handles 'set' magic. See `XPUSHn`.

```
void    PUSHn(double d)
```

PUSHp Push a string onto the stack. The stack must have room for this element. The `len` indicates the length of the string. Handles 'set' magic. See `XPUSHp`.

```
void    PUSHp(char *c, int len )
```

PUSHs Push an SV onto the stack. The stack must have room for this element. Does not handle 'set' magic. See `XPUSHs`.

```
void    PUSHs(sv)
```

PUSHu Push an unsigned integer onto the stack. The stack must have room for this element. See `XPUSHu`.

```
void    PUSHu(unsigned int d)
```

PUTBACK
Closing bracket for XSUB arguments. This is usually handled by `xsubpp`. See `PUSHMARK` and the *perlcall* manpage for other uses.

```
PUTBACK;
```

Renew The XSUB−writer's interface to the C `realloc` function.

```
void*   Renew( void *ptr, int size, type )
```

Renewc The XSUB−writer's interface to the C `realloc` function, with cast.

```
void*   Renewc( void *ptr, int size, type, cast )
```

RETVAL Variable which is setup by `xsubpp` to hold the return value for an XSUB. This is always the proper type for the XSUB. See the section on *The RETVAL Variable* in the *perlxs* manpage.

safefree
The XSUB−writer's interface to the C `free` function.

safemalloc
The XSUB−writer's interface to the C `malloc` function.

saferealloc
The XSUB−writer's interface to the C `realloc` function.

savepv
Copy a string to a safe spot. This does not use an SV.

```
char*   savepv (char* sv)
```

savepvn
Copy a string to a safe spot. The `len` indicates number of bytes to copy. This does not use an SV.

```
char*   savepvn (char* sv, I32 len)
```

SAVETMPS
Opening bracket for temporaries on a callback. See `FREETMPS` and the *perlcall* manpage.

```
SAVETMPS;
```

SP
Stack pointer. This is usually handled by `xsubpp`. See `dSP` and `SPAGAIN`.

SPAGAIN
Refetch the stack pointer. Used after a callback. See the *perlcall* manpage.

```
SPAGAIN;
```

ST
Used to access elements on the XSUB's stack.

```
SV*     ST(int x)
```

strEQ
Test two strings to see if they are equal. Returns true or false.

```
int     strEQ( char *s1, char *s2 )
```

strGE
Test two strings to see if the first, `s1`, is greater than or equal to the second, `s2`. Returns true or false.

```
int     strGE( char *s1, char *s2 )
```

strGT
Test two strings to see if the first, `s1`, is greater than the second, `s2`. Returns true or false.

```
int     strGT( char *s1, char *s2 )
```

strLE
Test two strings to see if the first, `s1`, is less than or equal to the second, `s2`. Returns true or false.

```
int     strLE( char *s1, char *s2 )
```

strLT
Test two strings to see if the first, `s1`, is less than the second, `s2`. Returns true or false.

```
int     strLT( char *s1, char *s2 )
```

strNE Test two strings to see if they are different. Returns true or false.

```
int     strNE( char *s1, char *s2 )
```

strnEQ Test two strings to see if they are equal. The `len` parameter indicates the number of bytes to compare. Returns true or false.

```
int     strnEQ( char *s1, char *s2 )
```

strnNE Test two strings to see if they are different. The `len` parameter indicates the number of bytes to compare. Returns true or false.

```
int     strnNE( char *s1, char *s2, int len )
```

sv_2mortal
Marks an SV as mortal. The SV will be destroyed when the current context ends.

```
SV*     sv_2mortal (SV* sv)
```

sv_bless Blesses an SV into a specified package. The SV must be an RV. The package must be designated by its stash (see `gv_stashpv()`). The reference count of the SV is unaffected.

```
SV*     sv_bless (SV* sv, HV* stash)
```

sv_catpv
Concatenates the string onto the end of the string which is in the SV. Handles 'get' magic, but not 'set' magic. See `sv_catpv_mg`.

```
void    sv_catpv (SV* sv, char* ptr)
```

sv_catpv_mg
Like `sv_catpv`, but also handles 'set' magic.

```
void    sv_catpv_mg (SV* sv, const char* ptr)
```

sv_catpvn
Concatenates the string onto the end of the string which is in the SV. The `len` indicates number of bytes to copy. Handles 'get' magic, but not 'set' magic. See `sv_catpvn_mg`.

```
void    sv_catpvn (SV* sv, char* ptr, STRLEN len)
```

sv_catpvn_mg
Like `sv_catpvn`, but also handles 'set' magic.

```
void    sv_catpvn_mg (SV* sv, char* ptr, STRLEN len)
```

sv_catpvf
Processes its arguments like `sprintf` and appends the formatted output to an SV. Handles 'get' magic, but not 'set' magic. `SvSETMAGIC()` must typically be called after calling this function to handle 'set' magic.

```
void    sv_catpvf (SV* sv, const char* pat, ...)
```

sv_catpvf_mg
Like `sv_catpvf`, but also handles 'set' magic.

```
void    sv_catpvf_mg (SV* sv, const char* pat, ...)
```

sv_catsv Concatenates the string from SV `ssv` onto the end of the string in SV `dsv`. Handles 'get' magic, but not 'set' magic. See `sv_catsv_mg`.

```
void    sv_catsv (SV* dsv, SV* ssv)
```

sv_catsv_mg
Like `sv_catsv`, but also handles 'set' magic.

```
void    sv_catsv_mg (SV* dsv, SV* ssv)
```

sv_chop Efficient removal of characters from the beginning of the string buffer. *SvPOK*(sv) must be true and the `ptr` must be a pointer to somewhere inside the string buffer. The `ptr` becomes the first character of the adjusted string.

```
void    sv_chop(SV* sv, char *ptr)
```

sv_cmp Compares the strings in two SVs. Returns −1, 0, or 1 indicating whether the string in `sv1` is less than, equal to, or greater than the string in `sv2`.

```
I32     sv_cmp (SV* sv1, SV* sv2)
```

SvCUR Returns the length of the string which is in the SV. See `SvLEN`.

```
int     SvCUR (SV* sv)
```

SvCUR_set
Set the length of the string which is in the SV. See `SvCUR`.

```
void    SvCUR_set (SV* sv, int val)
```

sv_dec Auto-decrement of the value in the SV.

```
void    sv_dec (SV* sv)
```

sv_derived_from
Returns a boolean indicating whether the SV is derived from the specified class. This is the function that implements `UNIVERSAL::isa`. It works for class names as well as for objects.

```
bool    sv_derived_from _((SV* sv, char* name));
```

SvEND Returns a pointer to the last character in the string which is in the SV. See `SvCUR`. Access the character as

```
char*   SvEND(sv)
```

sv_eq Returns a boolean indicating whether the strings in the two SVs are identical.

```
I32     sv_eq (SV* sv1, SV* sv2)
```

SvGETMAGIC
: Invokes `mg_get` on an SV if it has 'get' magic. This macro evaluates its argument more than once.

```
void    SvGETMAGIC(SV *sv)
```

SvGROW
: Expands the character buffer in the SV so that it has room for the indicated number of bytes (remember to reserve space for an extra trailing NUL character). Calls `sv_grow` to perform the expansion if necessary. Returns a pointer to the character buffer.

```
char*   SvGROW(SV* sv, STRLEN len)
```

sv_grow
: Expands the character buffer in the SV. This will use `sv_unref` and will upgrade the SV to `SVt_PV`. Returns a pointer to the character buffer. Use `SvGROW`.

sv_inc
: Auto-increment of the value in the SV.

```
void    sv_inc (SV* sv)
```

sv_insert
: Inserts a string at the specified offset/length within the SV. Similar to the Perl *substr()* function.

```
void    sv_insert(SV *sv, STRLEN offset, STRLEN len,
                  char *str, STRLEN strlen)
```

SvIOK
: Returns a boolean indicating whether the SV contains an integer.

```
int     SvIOK (SV* SV)
```

SvIOK_off
: Unsets the IV status of an SV.

```
void    SvIOK_off (SV* sv)
```

SvIOK_on
: Tells an SV that it is an integer.

```
void    SvIOK_on (SV* sv)
```

SvIOK_only
: Tells an SV that it is an integer and disables all other OK bits.

```
void    SvIOK_only (SV* sv)
```

SvIOKp
: Returns a boolean indicating whether the SV contains an integer. Checks the **private** setting. Use `SvIOK`.

```
int     SvIOKp (SV* SV)
```

sv_isa
: Returns a boolean indicating whether the SV is blessed into the specified class. This does not check for subtypes; use `sv_derived_from` to verify an inheritance relationship.

```
int     sv_isa (SV* sv, char* name)
```

sv_isobject
: Returns a boolean indicating whether the SV is an RV pointing to a blessed object. If the SV is not an RV, or if the object is not blessed, then this will return false.

```
int     sv_isobject (SV* sv)
```

SvIV
: Coerces the given SV to an integer and returns it.

```
int SvIV (SV* sv)
```

SvIVX
: Returns the integer which is stored in the SV, assuming SvIOK is true.

```
int     SvIVX (SV* sv)
```

SvLEN
: Returns the size of the string buffer in the SV. See `SvCUR`.

```
int     SvLEN (SV* sv)
```

sv_len
: Returns the length of the string in the SV. Use `SvCUR`.

```
STRLEN  sv_len (SV* sv)
```

sv_magic
: Adds magic to an SV.

```
void    sv_magic (SV* sv, SV* obj, int how, char* name, I32 namlen)
```

sv_mortalcopy
: Creates a new SV which is a copy of the original SV. The new SV is marked as mortal.

```
SV*     sv_mortalcopy (SV* oldsv)
```

sv_newmortal
: Creates a new SV which is mortal. The reference count of the SV is set to 1.

```
SV*     sv_newmortal (void)
```

SvNIOK
: Returns a boolean indicating whether the SV contains a number, integer or double.

```
int     SvNIOK (SV* SV)
```

SvNIOK_off
: Unsets the NV/IV status of an SV.

```
void    SvNIOK_off (SV* sv)
```

SvNIOKp
: Returns a boolean indicating whether the SV contains a number, integer or double. Checks the **private** setting. Use `SvNIOK`.

```
int     SvNIOKp (SV* SV)
```

PL_sv_no
: This is the `false` SV. See `PL_sv_yes`. Always refer to this as `&PL_sv_no`.

SvNOK
: Returns a boolean indicating whether the SV contains a double.

```
int     SvNOK (SV* SV)
```

SvNOK_off
: Unsets the NV status of an SV.

```
void    SvNOK_off (SV* sv)
```

SvNOK_on
: Tells an SV that it is a double.

```
void    SvNOK_on (SV* sv)
```

SvNOK_only
: Tells an SV that it is a double and disables all other OK bits.

```
void    SvNOK_only (SV* sv)
```

SvNOKp
: Returns a boolean indicating whether the SV contains a double. Checks the **private** setting. Use `SvNOK`.

```
int     SvNOKp (SV* SV)
```

SvNV
: Coerce the given SV to a double and return it.

```
double  SvNV (SV* sv)
```

SvNVX
: Returns the double which is stored in the SV, assuming SvNOK is true.

```
double  SvNVX (SV* sv)
```

SvOK
: Returns a boolean indicating whether the value is an SV.

```
int     SvOK (SV* sv)
```

SvOOK
: Returns a boolean indicating whether the SvIVX is a valid offset value for the SvPVX. This hack is used internally to speed up removal of characters from the beginning of a SvPV. When SvOOK is true, then the start of the allocated string buffer is really (SvPVX − SvIVX).

```
int     SvOOK(SV* sv)
```

SvPOK
: Returns a boolean indicating whether the SV contains a character string.

```
int     SvPOK (SV* SV)
```

SvPOK_off
: Unsets the PV status of an SV.

```
void    SvPOK_off (SV* sv)
```

SvPOK_on
Tells an SV that it is a string.

```
void    SvPOK_on (SV* sv)
```

SvPOK_only
Tells an SV that it is a string and disables all other OK bits.

```
void    SvPOK_only (SV* sv)
```

SvPOKp
Returns a boolean indicating whether the SV contains a character string. Checks the **private** setting. Use `SvPOK`.

```
int     SvPOKp (SV* SV)
```

SvPV
Returns a pointer to the string in the SV, or a stringified form of the SV if the SV does not contain a string. Handles 'get' magic.

```
char*   SvPV (SV* sv, STRLEN len)
```

SvPV_force
Like <SvPV> but will force the SV into becoming a string (SvPOK). You want force if you are going to update the SvPVX directly.

```
char*   SvPV_force(SV* sv, STRLEN len)
```

SvPVX
Returns a pointer to the string in the SV. The SV must contain a string.

```
char*   SvPVX (SV* sv)
```

SvREFCNT
Returns the value of the object's reference count.

```
int     SvREFCNT (SV* sv)
```

SvREFCNT_dec
Decrements the reference count of the given SV.

```
void    SvREFCNT_dec (SV* sv)
```

SvREFCNT_inc
Increments the reference count of the given SV.

```
void    SvREFCNT_inc (SV* sv)
```

SvROK
Tests if the SV is an RV.

```
int     SvROK (SV* sv)
```

SvROK_off
Unsets the RV status of an SV.

```
void    SvROK_off (SV* sv)
```

SvROK_on
Tells an SV that it is an RV.

```
void    SvROK_on (SV* sv)
```

SvRV
Dereferences an RV to return the SV.

```
SV*     SvRV (SV* sv)
```

SvSETMAGIC
Invokes `mg_set` on an SV if it has 'set' magic. This macro evaluates its argument more than once.

```
void    SvSETMAGIC( SV *sv )
```

sv_setiv
Copies an integer into the given SV. Does not handle 'set' magic. See `sv_setiv_mg`.

```
void    sv_setiv (SV* sv, IV num)
```

sv_setiv_mg
Like `sv_setiv`, but also handles 'set' magic.

```
void    sv_setiv_mg (SV* sv, IV num)
```

sv_setnv
Copies a double into the given SV. Does not handle 'set' magic. See `sv_setnv_mg`.

```
void    sv_setnv (SV* sv, double num)
```

sv_setnv_mg
Like `sv_setnv`, but also handles 'set' magic.

```
void    sv_setnv_mg (SV* sv, double num)
```

sv_setpv
Copies a string into an SV. The string must be null-terminated. Does not handle 'set' magic. See `sv_setpv_mg`.

```
void    sv_setpv (SV* sv, const char* ptr)
```

sv_setpv_mg
Like `sv_setpv`, but also handles 'set' magic.

```
void    sv_setpv_mg (SV* sv, const char* ptr)
```

sv_setpviv
Copies an integer into the given SV, also updating its string value. Does not handle 'set' magic. See `sv_setpviv_mg`.

```
void    sv_setpviv (SV* sv, IV num)
```

sv_setpviv_mg
Like `sv_setpviv`, but also handles 'set' magic.

```
void    sv_setpviv_mg (SV* sv, IV num)
```

sv_setpvn

Copies a string into an SV. The `len` parameter indicates the number of bytes to be copied. Does not handle 'set' magic. See `sv_setpvn_mg`.

```
void    sv_setpvn (SV* sv, const char* ptr, STRLEN len)
```

sv_setpvn_mg

Like `sv_setpvn`, but also handles 'set' magic.

```
void    sv_setpvn_mg (SV* sv, const char* ptr, STRLEN len)
```

sv_setpvf

Processes its arguments like `sprintf` and sets an SV to the formatted output. Does not handle 'set' magic. See `sv_setpvf_mg`.

```
void    sv_setpvf (SV* sv, const char* pat, ...)
```

sv_setpvf_mg

Like `sv_setpvf`, but also handles 'set' magic.

```
void    sv_setpvf_mg (SV* sv, const char* pat, ...)
```

sv_setref_iv

Copies an integer into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*     sv_setref_iv (SV *rv, char *classname, IV iv)
```

sv_setref_nv

Copies a double into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*     sv_setref_nv (SV *rv, char *classname, double nv)
```

sv_setref_pv

Copies a pointer into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. If the `pv` argument is NULL then `PL_sv_undef` will be placed into the SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*     sv_setref_pv (SV *rv, char *classname, void* pv)
```

Do not use with integral Perl types such as HV, AV, SV, CV, because those objects will become corrupted by the pointer copy process.

Note that `sv_setref_pvn` copies the string while this copies the pointer.

sv_setref_pvn

Copies a string into a new SV, optionally blessing the SV. The length of the string must be specified with `n`. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*     sv_setref_pvn (SV *rv, char *classname, char* pv, I32 n)
```

Note that `sv_setref_pv` copies the pointer while this copies the string.

SvSetSV
: Calls `sv_setsv` if dsv is not the same as ssv. May evaluate arguments more than once.

```
void    SvSetSV (SV* dsv, SV* ssv)
```

SvSetSV_nosteal
: Calls a non-destructive version of `sv_setsv` if dsv is not the same as ssv. May evaluate arguments more than once.

```
void    SvSetSV_nosteal (SV* dsv, SV* ssv)
```

sv_setsv
: Copies the contents of the source SV `ssv` into the destination SV `dsv`. The source SV may be destroyed if it is mortal. Does not handle 'set' magic. See the macro forms `SvSetSV`, `SvSetSV_nosteal` and `sv_setsv_mg`.

```
void    sv_setsv (SV* dsv, SV* ssv)
```

sv_setsv_mg
: Like `sv_setsv`, but also handles 'set' magic.

```
void    sv_setsv_mg (SV* dsv, SV* ssv)
```

sv_setuv
: Copies an unsigned integer into the given SV. Does not handle 'set' magic. See `sv_setuv_mg`.

```
void    sv_setuv (SV* sv, UV num)
```

sv_setuv_mg
: Like `sv_setuv`, but also handles 'set' magic.

```
void    sv_setuv_mg (SV* sv, UV num)
```

SvSTASH
: Returns the stash of the SV.

```
HV*     SvSTASH (SV* sv)
```

SvTAINT
: Taints an SV if tainting is enabled

```
void    SvTAINT (SV* sv)
```

SvTAINTED
: Checks to see if an SV is tainted. Returns TRUE if it is, FALSE if not.

```
int     SvTAINTED (SV* sv)
```

SvTAINTED_off
: Untaints an SV. Be *very* careful with this routine, as it short-circuits some of Perl's fundamental security features. XS module authors should not use this function unless they fully understand all

the implications of unconditionally untainting the value. Untainting should be done in the standard perl fashion, via a carefully crafted regexp, rather than directly untainting variables.

```
void    SvTAINTED_off (SV* sv)
```

SvTAINTED_on
Marks an SV as tainted.

```
void    SvTAINTED_on (SV* sv)
```

SVt_IV Integer type flag for scalars. See `svtype`.

SVt_PV Pointer type flag for scalars. See `svtype`.

SVt_PVAV
Type flag for arrays. See `svtype`.

SVt_PVCV
Type flag for code refs. See `svtype`.

SVt_PVHV
Type flag for hashes. See `svtype`.

SVt_PVMG
Type flag for blessed scalars. See `svtype`.

SVt_NV Double type flag for scalars. See `svtype`.

SvTRUE
Returns a boolean indicating whether Perl would evaluate the SV as true or false, defined or undefined. Does not handle 'get' magic.

```
int     SvTRUE (SV* sv)
```

SvTYPE
Returns the type of the SV. See `svtype`.

```
svtype  SvTYPE (SV* sv)
```

svtype An enum of flags for Perl types. These are found in the file **sv.h** in the `svtype` enum. Test these flags with the `SvTYPE` macro.

PL_sv_undef
This is the `undef` SV. Always refer to this as `&PL_sv_undef`.

sv_unref Unsets the RV status of the SV, and decrements the reference count of whatever was being referenced by the RV. This can almost be thought of as a reversal of `newSVrv`. See `SvROK_off`.

```
void    sv_unref (SV* sv)
```

SvUPGRADE
Used to upgrade an SV to a more complex form. Uses `sv_upgrade` to perform the upgrade if necessary. See `svtype`.

```
bool    SvUPGRADE (SV* sv, svtype mt)
```

sv_upgrade
Upgrade an SV to a more complex form. Use `SvUPGRADE`. See `svtype`.

sv_usepvn
Tells an SV to use `ptr` to find its string value. Normally the string is stored inside the SV but sv_usepvn allows the SV to use an outside string. The `ptr` should point to memory that was allocated by `malloc`. The string length, `len`, must be supplied. This function will realloc the memory pointed to by `ptr`, so that pointer should not be freed or used by the programmer after giving it to sv_usepvn. Does not handle 'set' magic. See `sv_usepvn_mg`.

```
void    sv_usepvn (SV* sv, char* ptr, STRLEN len)
```

sv_usepvn_mg
Like `sv_usepvn`, but also handles 'set' magic.

```
void    sv_usepvn_mg (SV* sv, char* ptr, STRLEN len)
```

sv_vcatpvfn(sv, pat, patlen, args, svargs, svmax, used_locale)
Processes its arguments like `vsprintf` and appends the formatted output to an SV. Uses an array of SVs if the C style variable argument list is missing (NULL). Indicates if locale information has been used for formatting.

```
void    sv_catpvfn _((SV* sv, const char* pat, STRLEN patlen,
                      va_list *args, SV **svargs, I32 svmax,
                      bool *used_locale));
```

sv_vsetpvfn(sv, pat, patlen, args, svargs, svmax, used_locale)
Works like `vcatpvfn` but copies the text into the SV instead of appending it.

```
void    sv_setpvfn _((SV* sv, const char* pat, STRLEN patlen,
                      va_list *args, SV **svargs, I32 svmax,
                      bool *used_locale));
```

SvUV Coerces the given SV to an unsigned integer and returns it.

```
UV      SvUV(SV* sv)
```

SvUVX Returns the unsigned integer which is stored in the SV, assuming SvIOK is true.

```
UV      SvUVX(SV* sv)
```

PL_sv_yes
This is the `true` SV. See `PL_sv_no`. Always refer to this as `&PL_sv_yes`.

THIS Variable which is setup by `xsubpp` to designate the object in a C++ XSUB. This is always the proper type for the C++ object. See `CLASS` and the section on *Using XS With C++* in the *perlxs* manpage.

toLOWER
Converts the specified character to lowercase.

```
int     toLOWER (char c)
```

toUPPER
Converts the specified character to uppercase.

```
int     toUPPER (char c)
```

warn
This is the XSUB−writer's interface to Perl's `warn` function. Use this function the same way you use the C `printf` function. See `croak()`.

XPUSHi
Push an integer onto the stack, extending the stack if necessary. Handles 'set' magic. See `PUSHi`.

```
XPUSHi(int d)
```

XPUSHn
Push a double onto the stack, extending the stack if necessary. Handles 'set' magic. See `PUSHn`.

```
XPUSHn(double d)
```

XPUSHp
Push a string onto the stack, extending the stack if necessary. The `len` indicates the length of the string. Handles 'set' magic. See `PUSHp`.

```
XPUSHp(char *c, int len)
```

XPUSHs
Push an SV onto the stack, extending the stack if necessary. Does not handle 'set' magic. See `PUSHs`.

```
XPUSHs(sv)
```

XPUSHu
Push an unsigned integer onto the stack, extending the stack if necessary. See `PUSHu`.

XS
Macro to declare an XSUB and its C parameter list. This is handled by `xsubpp`.

XSRETURN
Return from XSUB, indicating number of items on the stack. This is usually handled by `xsubpp`.

```
XSRETURN(int x)
```

XSRETURN_EMPTY
Return an empty list from an XSUB immediately.

```
XSRETURN_EMPTY;
```

XSRETURN_IV
Return an integer from an XSUB immediately. Uses `XST_mIV`.

```
XSRETURN_IV(IV v)
```

XSRETURN_NO
Return `&PL_sv_no` from an XSUB immediately. Uses `XST_mNO`.

```
XSRETURN_NO;
```

XSRETURN_NV
Return an double from an XSUB immediately. Uses `XST_mNV`.

```
XSRETURN_NV(NV v)
```

XSRETURN_PV

Return a copy of a string from an XSUB immediately. Uses `XST_mPV`.

```
XSRETURN_PV(char *v)
```

XSRETURN_UNDEF

Return `&PL_sv_undef` from an XSUB immediately. Uses `XST_mUNDEF`.

```
XSRETURN_UNDEF;
```

XSRETURN_YES

Return `&PL_sv_yes` from an XSUB immediately. Uses `XST_mYES`.

```
XSRETURN_YES;
```

XST_mIV

Place an integer into the specified position `i` on the stack. The value is stored in a new mortal SV.

```
XST_mIV( int i, IV v )
```

XST_mNV

Place a double into the specified position `i` on the stack. The value is stored in a new mortal SV.

```
XST_mNV( int i, NV v )
```

XST_mNO

Place `&PL_sv_no` into the specified position `i` on the stack.

```
XST_mNO( int i )
```

XST_mPV

Place a copy of a string into the specified position `i` on the stack. The value is stored in a new mortal SV.

```
XST_mPV( int i, char *v )
```

XST_mUNDEF

Place `&PL_sv_undef` into the specified position `i` on the stack.

```
XST_mUNDEF( int i )
```

XST_mYES

Place `&PL_sv_yes` into the specified position `i` on the stack.

```
XST_mYES( int i )
```

XS_VERSION

The version identifier for an XS module. This is usually handled automatically by `ExtUtils::MakeMaker`. See `XS_VERSION_BOOTCHECK`.

XS_VERSION_BOOTCHECK

Macro to verify that a PM module's `$VERSION` variable matches the XS module's `XS_VERSION` variable. This is usually handled automatically by `xsubpp`. See the section on *The VERSIONCHECK: Keyword* in the *perlxs* manpage.

Zero
: The XSUB−writer's interface to the C `memzero` function. The `d` is the destination, `n` is the number of items, and `t` is the type.

```
void    Zero( d, n, t )
```

AUTHORS

Until May 1997, this document was maintained by Jeff Okamoto <okamoto@corp.hp.com>. It is now maintained as part of Perl itself.

With lots of help and suggestions from Dean Roehrich, Malcolm Beattie, Andreas Koenig, Paul Hudson, Ilya Zakharevich, Paul Marquess, Neil Bowers, Matthew Green, Tim Bunce, Spider Boardman, Ulrich Pfeifer, Stephen McCamant, and Gurusamy Sarathy.

API Listing originally by Dean Roehrich <roehrich@cray.com>.