NAME

perlguts - Introduction to the Perl API

DESCRIPTION

This document attempts to describe how to use the Perl API, as well as to provide some info on the basic workings of the Perl core. 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 signed integer type that is guaranteed to be large enough to hold a pointer (as well as an integer). Additionally, there is the UV, which is simply an unsigned IV.

Perl also uses two special typedefs, I32 and I16, which will always be at least 32-bits and 16-bits long, respectively. (Again, there are U32 and U16, as well.) They will usually be exactly 32 and 16 bits long, but on Crays they will both be 64 bits.

Working with SVs

An SV can be created and loaded with one command. There are five types of values that can be loaded: an integer value (IV), an unsigned integer value (UV), a double (NV), a string (PV), and another scalar (SV). ("PV" stands for "Pointer Value". You might think that it is misnamed because it is described as pointing only to strings. However, it is possible to have it point to other things. For example, it could point to an array of UVs. But, using it for non-strings requires care, as the underlying assumption of much of the internals is that PVs are just for strings. Often, for example, a trailing `NUL` is tacked on automatically. The non-string use is documented only in this paragraph.)

The seven routines are:

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

`STRLEN` is an integer type (Size_t, usually defined as size_t in *config.h*) guaranteed to be large enough to represent the size of any string that perl can handle.

In the unlikely case of a SV requiring more complex initialization, you can create an empty SV with newSV(len). If `len` is 0 an empty SV of type NULL is returned, else an SV of type PV is returned with len + 1 (for the `NUL`) bytes of storage allocated, accessible via SvPVX. In both cases the SV has the undef value.

```
SV *sv = newSV(0);   /* no storage allocated  */
SV *sv = newSV(10);  /* 10 (+1) bytes of uninitialised storage
                      * allocated */
```

To change the value of an *already-existing* SV, there are eight 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*, STRLEN)
void  sv_setpvf(SV*, const char*, ...);
void  sv_vsetpvfn(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, and not otherwise containing NULs.

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

`sv_vsetpvfn` 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 *perlsec*). 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 *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*)
SvUV(SV*)
SvNV(SV*)
SvPV(SV*, STRLEN len)
SvPV_nolen(SV*)
```

which will automatically coerce the actual scalar type into an IV, UV, 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 `SvPV_nolen` macro. Historically the `SvPV` macro with the global variable `PL_na` has been used in this case. But that 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:

```
SV *s;
STRLEN len;
```

```
    char *ptr;
    ptr = SvPV(s, 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 space for the trailing `NUL` byte (perl's own string functions typically do `SvGROW(sv, len + 1)`).

If you want to write to an existing SV's buffer and set its value to a string, use SvPV_force() or one of its variants to force the SV to be a PV. This will remove any of various types of non-stringness from the SV while preserving the content of the SV in the PV. This can be used, for example, to append data from an API function to a buffer without extra copying:

```
    (void)SvPVbyte_force(sv, len);
    s = SvGROW(sv, len + needlen + 1);
    /* something that modifies up to needlen bytes at s+len, but
       modifies newlen bytes
         eg. newlen = read(fd, s + len, needlen);
       ignoring errors for these examples
     */
    s[len + newlen] = '\0';
    SvCUR_set(sv, len + newlen);
    SvUTF8_off(sv);
    SvSETMAGIC(sv);
```

If you already have the data in memory or if you want to keep your code simple, you can use one of the sv_cat*() variants, such as sv_catpvn(). If you want to insert anywhere in the string you can use sv_insert() or sv_insert_flags().

If you don't need the existing content of the SV, you can avoid some copying with:

```
    SvPVCLEAR(sv);
    s = SvGROW(sv, needlen + 1);
    /* something that modifies up to needlen bytes at s, but modifies
       newlen bytes
         eg. newlen = read(fd, s. needlen);
     */
    s[newlen] = '\0';
    SvCUR_set(sv, newlen);
    SvPOK_only(sv); /* also clears SVf_UTF8 */
    SvSETMAGIC(sv);
```

Again, if you already have the data in memory or want to avoid the complexity of the above, you can use sv_setpvn().

If you have a buffer allocated with Newx() and want to set that as the SV's value, you can use sv_usepvn_flags(). That has some requirements if you want to avoid perl re-allocating the buffer to fit

the trailing NUL:

```
Newx(buf, somesize+1, char);
/* ... fill in buf ... */
buf[somesize] = '\0';
sv_usepvn_flags(sv, buf, somesize, SV_SMAGIC | SV_HAS_TRAILING_NUL);
/* buf now belongs to perl, don't release it */
```

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*, const char*);
void  sv_catpvn(SV*, const char*, STRLEN);
void  sv_catpvf(SV*, const char*, ...);
void  sv_vcatpvfn(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 *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*  get_sv("package::varname", 0);
```

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. Make sure that you don't try to compare a random sv with &PL_sv_undef. For example when interfacing Perl code, it'll work correctly for:

```
foo(undef);
```

But won't work when called as:

```
$x = undef;
foo($x);
```

So to repeat always use SvOK() to check whether an sv is defined.

Also you have to be careful when using &PL_sv_undef as a value in AVs or HVs (see *AVs, HVs and undefined values*).

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 *Reference Counts and Mortality*).

Offsets

Perl provides the function sv_chop to efficiently remove characters from the beginning of a string; you give it an SV and a pointer to somewhere inside the PV, and it discards everything before the pointer. The efficiency comes by means of a little hack: instead of actually removing the characters, sv_chop sets the flag OOK (offset OK) to signal to other functions that the offset hack is in effect, and it moves the PV pointer (called SvPVX) forward by the number of bytes chopped off, and adjusts SvCUR and SvLEN accordingly. (A portion of the space between the old and new PV pointers is used to store the count of chopped bytes.)

Hence, at this point, the start of the buffer that we allocated lives at SvPVX(sv) - SvIV(sv) in memory and the PV pointer is pointing into the middle of this allocated storage.

This is best demonstrated by example. Normally copy-on-write will prevent the substitution from operator from using this hack, but if you can craft a string for which copy-on-write is not possible, you can see it in play. In the current implementation, the final byte of a string buffer is used as a copy-on-write reference count. If the buffer is not big enough, then copy-on-write is skipped. First have a look at an empty string:

```
% ./perl -Ilib -MDevel::Peek -le '$a=""; $a .= ""; Dump $a'
SV = PV(0x7ffb7c008a70) at 0x7ffb7c030390
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x7ffb7bc05b50 ""\0
  CUR = 0
```

```
    LEN = 10
```

Notice here the LEN is 10. (It may differ on your platform.) Extend the length of the string to one less than 10, and do a substitution:

```
 % ./perl -Ilib -MDevel::Peek -le '$a=""; $a.="123456789"; $a=~s/.//; \
                                                          Dump($a)'
 SV = PV(0x7ffa04008a70) at 0x7ffa04030390
   REFCNT = 1
   FLAGS = (POK,OOK,pPOK)
   OFFSET = 1
   PV = 0x7ffa03c05b61 ( "\1" . ) "23456789"\0
   CUR = 8
   LEN = 9
```

Here the number of bytes chopped off (1) is shown next as the OFFSET. The portion of the string between the "real" and the "fake" beginnings is shown in parentheses, and the values of `SvCUR` and `SvLEN` reflect the fake beginning, not the real one. (The first character of the string buffer happens to have changed to "\1" here, not "1", because the current implementation stores the offset count in the string buffer. This is subject to change.)

Something similar to the offset hack is performed on AVs to enable efficient shifting and splicing off the beginning of the array; while `AvARRAY` points to the first element in the array that is visible from Perl, `AvALLOC` points to the real start of the C array. These are usually the same, but a `shift` operation can be carried out by increasing `AvARRAY` by one and decreasing `AvFILL` and `AvMAX`. Again, the location of the real start of the C array only comes into play when freeing the array. See `av_shift` in *av.c*.

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.

There are various ways in which the private and public flags may differ. For example, in perl 5.16 and earlier a tied SV may have a valid underlying value in the IV slot (so SvIOKp is true), but the data should be accessed via the FETCH routine rather than directly, so SvIOK is false. (In perl 5.18 onwards, tied scalars use the flags the same way as untied scalars.) Another is when numeric conversion has occurred and precision has been lost: only the private flag is set on 'lossy' values. So when an NV is converted to an IV with loss, SvIOKp, SvNOKp and SvNOK will be set, while SvIOK wont be.

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(SSize_t 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 it:

```
void  av_push(AV*, SV*);
SV*   av_pop(AV*);
SV*   av_shift(AV*);
void  av_unshift(AV*, SSize_t 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:

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

The `av_top_index` function returns the highest index value in an 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.

A few more:

```
void  av_clear(AV*);
void  av_undef(AV*);
void  av_extend(AV*, SSize_t 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*  get_av("package::varname", 0);
```

This returns NULL if the variable does not exist.

See *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 it:

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

The first of these two functions checks if a hash table entry exists, and the second deletes it.

```
bool  hv_exists(HV*, const char* key, U32 klen);
SV*   hv_delete(HV*, const 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 a 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 an 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*  get_hv("package::varname", 0);
```

This returns NULL if the variable does not exist.

The hash algorithm is defined in the PERL_HASH macro:

```
PERL_HASH(hash, key, klen)
```

The exact implementation of this macro varies by architecture and version of perl, and the return value may change per invocation, so the value is only valid for the duration of a single perl process.

See *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 *perlapi* 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 *perlapi* 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.

AVs, HVs and undefined values

Sometimes you have to store undefined values in AVs or HVs. Although this may be a rare case, it can be tricky. That's because you're used to using &PL_sv_undef if you need an undefined SV.

For example, intuition tells you that this XS code:

```
AV *av = newAV();
av_store( av, 0, &PL_sv_undef );
```

is equivalent to this Perl code:

```
my @av;
$av[0] = undef;
```

Unfortunately, this isn't true. In perl 5.18 and earlier, AVs use `&PL_sv_undef` as a marker for indicating that an array element has not yet been initialized. Thus, `exists $av[0]` would be true for the above Perl code, but false for the array generated by the XS code. In perl 5.20, storing &PL_sv_undef will create a read-only element, because the scalar &PL_sv_undef itself is stored, not a copy.

Similar problems can occur when storing `&PL_sv_undef` in HVs:

```
hv_store( hv, "key", 3, &PL_sv_undef, 0 );
```

This will indeed make the value `undef`, but if you try to modify the value of `key`, you'll get the following error:

```
Modification of non-creatable hash value attempted
```

In perl 5.8.0, `&PL_sv_undef` was also used to mark placeholders in restricted hashes. This caused such hash entries not to appear when iterating over the hash or when checking for the keys with the `hv_exists` function.

You can run into similar problems when you store `&PL_sv_yes` or `&PL_sv_no` into AVs or HVs. Trying to modify such elements will give you the following error:

```
Modification of a read-only value attempted
```

To make a long story short, you can use the special variables `&PL_sv_undef`, `&PL_sv_yes` and `&PL_sv_no` with AVs and HVs, but you have to make sure you know what you're doing.

Generally, if you want to store an undefined value in an AV or HV, you should not use `&PL_sv_undef`, but rather create a new undefined value using the `newSV` function, for example:

```
av_store( av, 42, newSV(0) );
hv_store( hv, "foo", 3, newSV(0), 0 );
```

References

References are a special type of scalar that point to other data types (including other 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_PVAV  Scalar
SVt_PVAV    Array
SVt_PVHV    Hash
SVt_PVCV    Code
SVt_PVGV    Glob (possibly a file handle)
```

See *"svtype" in perlapi* for more details.

Blessed References and Class Objects

References are also used to support object-oriented programming. In perl's 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 value. The stash argument specifies which class the reference will belong to. See *Stashes and Globs* for information on converting class names into stashes.

/* Still under construction */

The following function upgrades rv to reference if not already one. Creates a 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, const char* classname);
```

The following three functions copy integer, unsigned integer or double into an SV whose reference is rv. SV is blessed if classname is non-null.

```
SV* sv_setref_iv(SV* rv, const char* classname, IV iv);
SV* sv_setref_uv(SV* rv, const char* classname, UV uv);
SV* sv_setref_nv(SV* rv, const char* classname, NV iv);
```

The following function 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, const char* classname, void* pv);
```

The following function copies a 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, const char* classname, char* pv,
                                                 STRLEN length);
```

The following function tests whether the SV is blessed into the specified class. It does not check inheritance relationships.

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

The following function tests whether the SV is a reference to a blessed object.

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

The following function 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, const 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*  get_sv("package::varname", GV_ADD);
AV*  get_av("package::varname", GV_ADD);
HV*  get_hv("package::varname", GV_ADD);
```

Notice the use of GV_ADD 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 `GV_ADD` 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 a 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 *perlcall* and *perlxs* 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.

"Mortal" SVs are mainly used for SVs that are placed on perl's stack. For example an SV which is created just to pass a number to a called sub is made mortal to have it cleaned up automatically when it's popped off the stack. Similarly, results returned by XSUBs (which are pushed on the stack) are often made mortal.

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 (with no value), 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. Because sv_newmortal gives the new SV no value, it must normally be given one via sv_setpv, sv_setiv, etc. :

```
SV *tmp = sv_newmortal();
sv_setiv(tmp, an_integer);
```

As that is multiple C statements it is quite common so see this idiom instead:

```
SV *tmp = sv_2mortal(newSViv(an_integer));
```

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. Thinking of "Mortalization" as deferred SvREFCNT_dec should help to minimize such problems. For example if you are passing an SV which you *know* has a high enough REFCNT to survive its use on the stack you need not do any mortalization. If you are not sure then doing an SvREFCNT_inc and

`sv_2mortal`, or making a `sv_mortalcopy` is safer.

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 variables that are defined 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(const char* name, I32 flags)
HV*  gv_stashsv(SV*, I32 flags)
```

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 `flags` flag will create a new package if it is set to GV_ADD.

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 *perlref*.

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 = get_sv("dberror", GV_ADD);
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`.

Read-Only Values

In Perl 5.16 and earlier, copy-on-write (see the next section) shared a flag bit with read-only scalars. So the only way to test whether `sv_setsv`, etc., will raise a "Modification of a read-only value" error in those versions is:

```
SvREADONLY(sv) && !SvIsCOW(sv)
```

Under Perl 5.18 and later, SvREADONLY only applies to read-only variables, and, under 5.20, copy-on-write scalars can also be read-only, so the above check is incorrect. You just want:

```
SvREADONLY(sv)
```

If you need to do this check often, define your own macro like this:

```
#if PERL_VERSION >= 18
# define SvTRULYREADONLY(sv) SvREADONLY(sv)
#else
# define SvTRULYREADONLY(sv) (SvREADONLY(sv) && !SvIsCOW(sv))
#endif
```

Copy on Write

Perl implements a copy-on-write (COW) mechanism for scalars, in which string copies are not immediately made when requested, but are deferred until made necessary by one or the other scalar changing. This is mostly transparent, but one must take care not to modify string buffers that are shared by multiple SVs.

You can test whether an SV is using copy-on-write with `SvIsCOW(sv)`.

You can force an SV to make its own copy of its string buffer by calling `sv_force_normal(sv)` or SvPV_force_nolen(sv).

If you want to make the SV drop its string buffer, use `sv_force_normal_flags(sv, SV_COW_DROP_PV)` or simply `sv_setsv(sv, NULL)`.

All of these functions will croak on read-only scalars (see the previous section for more on those).

To test that your code is behaving correctly and not modifying COW buffers, on systems that support *mmap(2)* (i.e., Unix) you can configure perl with `-Accflags=-DPERL_DEBUG_READONLY_COW` and it will turn buffer violations into crashes. You will find it to be marvellously slow, so you may want to skip perl's own tests.

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;
    I32         mg_len;
    SV*         mg_obj;
    char*       mg_ptr;
};
```

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, const 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 convert `sv` to type `SVt_PVMG`. Perl then continues by adding new magic 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 then either a `savepvn` copy of `name` or `name` itself is stored in the `mg_ptr` field, depending on whether `namlen` is greater than zero or equal to zero respectively. As a special case, if `(name && namlen == HEf_SVKEY)` then `name` is assumed to contain an `SV*` and is stored as-is with its REFCNT incremented.

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 Tables* section below. The `how` argument is also stored in the `mg_type` field. The value of `how` should be chosen from the set of macros `PERL_MAGIC_foo` found in *perl.h*. Note that before these macros were added, Perl internals used to directly use character literals, so you may occasionally come across old code or documentation referring to 'U' magic rather than `PERL_MAGIC_uvar` for example.

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 PERL_MAGIC_arylen, PERL_MAGIC_regdatum, PERL_MAGIC_regdata, or if it is a NULL pointer, then obj is merely stored, without the reference count being incremented.

See also sv_magicext in *perlapi* for a more flexible way to add magic to an SV.

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:

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

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

However, note that sv_unmagic removes all magic of a certain type from the SV. If you want to remove only certain magic of a type based on the magic virtual table, use sv_unmagicext instead:

```
int sv_unmagicext(SV *sv, int type, MGVTBL *vtbl);
```

Magic Virtual Tables

The mg_virtual field in the MAGIC structure is a pointer to an 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 (or sometimes eight) pointers to the following routine types:

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

int  (*svt_copy) (pTHX_ SV *sv, MAGIC* mg, SV *nsv,
                                    const char *name, I32 namlen);
int  (*svt_dup)  (pTHX_ MAGIC *mg, CLONE_PARAMS *param);
int  (*svt_local)(pTHX_ SV *nsv, MAGIC *mg);
```

This MGVTBL structure is set at compile-time in *perl.h* and there are currently 32 types. 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 before 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.

svt_copy            copy tied variable magic to a tied element
svt_dup             duplicate a magic structure during thread cloning
```

```
    svt_local           copy magic to local value during 'local'
```

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

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

Thus, when an SV is determined to be magical and of type `PERL_MAGIC_sv`, 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_`. NOTE: the magic routines are not considered part of the Perl API, and may not be exported by the Perl library.

The last three slots are a recent addition, and for source code compatibility they are only checked for if one of the three flags MGf_COPY, MGf_DUP or MGf_LOCAL is set in mg_flags. This means that most code can continue declaring a vtable as a 5-element value. These three are currently used exclusively by the threading code, and are highly subject to change.

The current kinds of Magic Virtual Tables are:

```
mg_type
(old-style char and macro)   MGVTBL         Type of magic
--------------------------   ------         -------------
\0 PERL_MAGIC_sv             vtbl_sv        Special scalar variable
#  PERL_MAGIC_arylen         vtbl_arylen    Array length ($#ary)
%  PERL_MAGIC_rhash          (none)         Extra data for restricted
                                            hashes
*  PERL_MAGIC_debugvar       vtbl_debugvar  $DB::single, signal, trace
                                            vars
.  PERL_MAGIC_pos            vtbl_pos       pos() lvalue
:  PERL_MAGIC_symtab         (none)         Extra data for symbol
                                            tables
<  PERL_MAGIC_backref        vtbl_backref   For weak ref data
@  PERL_MAGIC_arylen_p       (none)         To move arylen out of XPVAV
B  PERL_MAGIC_bm             vtbl_regexp    Boyer-Moore
                                            (fast string search)
c  PERL_MAGIC_overload_table vtbl_ovrld     Holds overload table
                                            (AMT) on stash
D  PERL_MAGIC_regdata        vtbl_regdata   Regex match position data
                                            (@+ and @- vars)
d  PERL_MAGIC_regdatum       vtbl_regdatum  Regex match position data
                                            element
E  PERL_MAGIC_env            vtbl_env       %ENV hash
e  PERL_MAGIC_envelem        vtbl_envelem   %ENV hash element
f  PERL_MAGIC_fm             vtbl_regexp    Formline
                                            ('compiled' format)
g  PERL_MAGIC_regex_global   vtbl_mglob     m//g target
H  PERL_MAGIC_hints          vtbl_hints     %^H hash
h  PERL_MAGIC_hintselem      vtbl_hintselem %^H hash element
I  PERL_MAGIC_isa            vtbl_isa       @ISA array
i  PERL_MAGIC_isaelem        vtbl_isaelem   @ISA array element
k  PERL_MAGIC_nkeys          vtbl_nkeys     scalar(keys()) lvalue
L  PERL_MAGIC_dbfile         (none)         Debugger %_<filename
l  PERL_MAGIC_dbline         vtbl_dbline    Debugger %_<filename
                                            element
N  PERL_MAGIC_shared         (none)         Shared between threads
n  PERL_MAGIC_shared_scalar  (none)         Shared between threads
o  PERL_MAGIC_collxfrm       vtbl_collxfrm  Locale transformation
```

```
P  PERL_MAGIC_tied           vtbl_pack      Tied array or hash
p  PERL_MAGIC_tiedelem       vtbl_packelem  Tied array or hash element
q  PERL_MAGIC_tiedscalar     vtbl_packelem  Tied scalar or handle
r  PERL_MAGIC_qr             vtbl_regexp    Precompiled qr// regex
S  PERL_MAGIC_sig            (none)         %SIG hash
s  PERL_MAGIC_sigelem        vtbl_sigelem   %SIG hash element
t  PERL_MAGIC_taint          vtbl_taint     Taintedness
U  PERL_MAGIC_uvar           vtbl_uvar      Available for use by
                                            extensions
u  PERL_MAGIC_uvar_elem      (none)         Reserved for use by
                                            extensions
V  PERL_MAGIC_vstring        (none)         SV was vstring literal
v  PERL_MAGIC_vec            vtbl_vec       vec() lvalue
w  PERL_MAGIC_utf8           vtbl_utf8      Cached UTF-8 information
x  PERL_MAGIC_substr         vtbl_substr    substr() lvalue
y  PERL_MAGIC_defelem        vtbl_defelem   Shadow "foreach" iterator
                                            variable / smart parameter
                                            vivification
\  PERL_MAGIC_lvref          vtbl_lvref     Lvalue reference
                                            constructor
]  PERL_MAGIC_checkcall      vtbl_checkcall Inlining/mutation of call
                                            to this CV
~  PERL_MAGIC_ext            (none)         Available for use by
                                            extensions
```

When an uppercase and lowercase letter both exist in the table, then the uppercase letter is typically 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. Some internals code makes use of this case relationship. However, 'v' and 'V' (vec and v-string) are in no way related.

The `PERL_MAGIC_ext` and `PERL_MAGIC_uvar` magic types are defined specifically for use by extensions and will not be used by perl itself. Extensions can use `PERL_MAGIC_ext` 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, `PERL_MAGIC_uvar` 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)(pTHX_ IV, SV*);
    I32 (*uf_set)(pTHX_ 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 `PERL_MAGIC_uvar` 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, PERL_MAGIC_uvar, (char*)&uf, sizeof(uf));
```

Attaching `PERL_MAGIC_uvar` to arrays is permissible but has no effect.

For hashes there is a specialized hook that gives control over hash keys (but not values). This hook calls `PERL_MAGIC_uvar` 'get' magic if the "set" function in the `ufuncs` structure is NULL. The hook is activated whenever the hash is accessed with a key specified as an SV through the functions `hv_store_ent`, `hv_fetch_ent`, `hv_delete_ent`, and `hv_exists_ent`. Accessing the key as a string through the functions without the `..._ent` suffix circumvents the hook. See *"GUTS" in Hash::Util::FieldHash* for a detailed description.

Note that because multiple extensions may be using `PERL_MAGIC_ext` or `PERL_MAGIC_uvar` 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 `PERL_MAGIC_ext` magic, it is usually a good idea to define an `MGVTBL`, even if all its fields will be `0`, so that individual `MAGIC` pointers can be identified as a particular kind of magic using their magic virtual table. `mg_findext` provides an easy way to do that:

```
STATIC MGVTBL my_vtbl = { 0, 0, 0, 0, 0, 0, 0, 0 };

MAGIC *mg;
if ((mg = mg_findext(sv, PERL_MAGIC_ext, &my_vtbl))) {
    /* this is really ours, not another module's PERL_MAGIC_ext */
    my_priv_data_t *priv = (my_priv_data_t *)mg->mg_ptr;
    ...
}
```

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. See *perlapi* for a description of these 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 *sv, int type); /* Finds the magic pointer of that
                                   * type */
```

This routine returns a pointer to a `MAGIC` structure stored in the SV. If the SV does not have that magical feature, `NULL` is returned. If the SV has multiple instances of that magical feature, the first one will be returned. `mg_findext` can be used to find a `MAGIC` structure of an SV based on both its magic type and its magic virtual table:

```
MAGIC *mg_findext(SV *sv, int type, MGVTBL *vtbl);
```

Also, if the SV passed to `mg_find` or `mg_findext` is not of type SVt_PVMG, Perl may core dump.

```
int mg_copy(SV* sv, SV* nsv, const 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 `PERL_MAGIC_tied` 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 *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", GV_ADD);
    sv_bless(tie, stash);
    hv_magic(hash, (GV*)tie, PERL_MAGIC_tied);
    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 *"Returning a Scalar" in perlcall*). 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` will be decremented at the end of *pseudo-block*. This is similar to

sv_2mortal in that it is also a mechanism for doing a delayed SvREFCNT_dec. However, while sv_2mortal extends the lifetime of sv until the beginning of the next statement, SAVEFREESV extends it until the end of the enclosing scope. These lifetimes can be wildly different.

Also compare SAVEMORTALIZESV.

SAVEMORTALIZESV(SV *sv)

Just like SAVEFREESV, but mortalizes sv at the end of the current scope instead of decrementing its reference count. This usually has the effect of keeping sv alive until the statement that called the currently live scope has finished executing.

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(DESTRUCTORFUNC_NOCONTEXT_t f, void *p)

At the end of *pseudo-block* the function f is called with the only argument p.

SAVEDESTRUCTOR_X(DESTRUCTORFUNC_t f, void *p)

At the end of *pseudo-block* the function f is called with the implicit context argument (if any), and 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. It doesn't handle magic. Use save_scalar if magic is affected.

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 an `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 `PUSHs` macro. The pushed values will often need to be "mortal" (See *Reference Counts and Mortality*):

```
PUSHs(sv_2mortal(newSViv(an_integer)))
PUSHs(sv_2mortal(newSVuv(an_unsigned_integer)))
PUSHs(sv_2mortal(newSVnv(a_double)))
PUSHs(sv_2mortal(newSVpv("Some String",0)))
/* Although the last example is better written as the more
 * efficient: */
PUSHs(newSVpvs_flags("Some String", SVs_TEMP))
```

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 macro:

```
XPUSHs(SV*)
```

This macro automatically adjusts the stack for you, if needed. Thus, you do not need to call `EXTEND` to extend the stack.

Despite their suggestions in earlier versions of this document the macros `(X)PUSH[iunp]` are *not* suited to XSUBs which return multiple results. For that, either stick to the `(X)PUSHs` macros shown above, or use the new `m(X)PUSH[iunp]` macros instead; see *Putting a C value on Perl stack*.

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

Autoloading with XSUBs

If an AUTOLOAD routine is an XSUB, as with Perl subroutines, Perl puts the fully-qualified name of the autoloaded subroutine in the $AUTOLOAD variable of the XSUB's package.

But it also puts the same information in certain fields of the XSUB itself:

```
HV *stash           = CvSTASH(cv);
const char *subname = SvPVX(cv);
STRLEN name_length  = SvCUR(cv); /* in bytes */
U32 is_utf8         = SvUTF8(cv);
```

`SvPVX(cv)` contains just the sub name itself, not including the package. For an AUTOLOAD routine in UNIVERSAL or one of its superclasses, `CvSTASH(cv)` returns NULL during a method call on a nonexistent package.

Note: Setting $AUTOLOAD stopped working in 5.6.1, which did not support XS AUTOLOAD subs at all. Perl 5.8.0 introduced the use of fields in the XSUB itself. Perl 5.16.0 restored the setting of $AUTOLOAD. If you need to support 5.8-5.14, use the XSUB's fields.

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  call_sv(SV*, I32);
I32  call_pv(const char*, I32);
I32  call_method(const char*, I32);
I32  call_argv(const char*, I32, char**);
```

The routine most often used is `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.

These routines used to be called `perl_call_sv`, etc., before Perl v5.6.0, but those names are now deprecated; macros of the same name are provided for compatibility.

When using any of these routines (except `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 *perlcall*.

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 *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[iunp]`.

Because the target is reused, you must be careful when pushing multiple values on the stack. The following code will not do what you think:

```
XPUSHi(10);
XPUSHi(20);
```

This translates as "set `TARG` to 10, push a pointer to `TARG` onto the stack; set `TARG` to 20, push a pointer to `TARG` onto the stack". At the end of the operation, the stack does not contain the values 10 and 20, but actually contains two pointers to `TARG`, which we have set to 20.

If you need to push multiple different values then you should either use the `(X)PUSHs` macros, or else use the new `m(X)PUSH[iunp]` macros, none of which make use of `TARG`. The `(X)PUSHs` macros simply push an SV* on the stack, which, as noted under *XSUBs and the Argument Stack*, will often need to be "mortal". The new `m(X)PUSH[iunp]` macros make this a little easier to achieve by creating a new mortal for you (via `(X)PUSHmortal`), pushing that onto the stack (extending it if necessary in the case of the `mXPUSH[iunp]` macros), and then setting its value. Thus, instead of writing this to "fix" the example above:

```
XPUSHs(sv_2mortal(newSViv(10)))
XPUSHs(sv_2mortal(newSViv(20)))
```

you can simply write:

```
mXPUSHi(10)
mXPUSHi(20)
```

On a related note, if you do use `(X)PUSH[iunp]`, then you're going to need a `dTARG` in your variable declarations so that the `*PUSH*` macros can make use of the local variable `TARG`. See also `dTARGET` and `dXSTARG`.

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. A previous version of this document stated that 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. But this has never been fully true. `SVs_PADMY` could be set on a variable that no longer resides in any pad. While *target*s do have `SVs_PADTMP` set, it can also be set on variables that have never resided in a pad, but nonetheless act like *target*s. As of perl 5.21.5, the `SVs_PADMY` flag is no longer used and is defined as 0. `SvPADMY()` now returns true for anything without `SVs_PADTMP`.

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 **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 `undefs`, but they are already marked with correct flags.

Memory Allocation

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.

The following three macros are used to initially allocate memory :

```
Newx(pointer, number, type);
Newxc(pointer, number, type, cast);
Newxz(pointer, number, type);
```

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

The second and third 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 `Newxc`, `cast`, should be used if the `pointer` argument is different from the `type` argument.

Unlike the `Newx` and `Newxc` macros, the `Newxz` macro calls `memzero` to zero out all the newly allocated memory.

Reallocation

```
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.

Moving

```
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 have 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 *perlapio*.

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 `-DDEBUGGING` on the `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`.

Each of these nodes represents an op, a fundamental operation inside the Perl core. The code which implements each operation can be found in the *pp*.c* files; the function which implements the op with type `gvsv` is `pp_gvsv`, and so on. As the tree above shows, different ops have different numbers of children: `add` is a binary operator, as one would expect, and so has two children. To accommodate the various different numbers of children, there are various types of op data structure, and they link together in different ways.

The simplest type of op structure is `OP`: this has no children. Unary operators, `UNOP`s, have one child, and this is pointed to by the `op_first` field. Binary operators (`BINOP`s) have not only an `op_first` field but also an `op_last` field. The most complex type of op is a `LISTOP`, which has any number of children. In this case, the first child is pointed to by `op_first` and the last child by `op_last`. The children in between can be found by iteratively following the `OpSIBLING` pointer from the first child to the last (but see below).

There are also some other op types: a `PMOP` holds a regular expression, and has no children, and a `LOOP` may or may not have children. If the `op_children` field is non-zero, it behaves like a `LISTOP`. To complicate matters, if a `UNOP` is actually a `null` op after optimization (see *Compile pass 2: context propagation*) it will still have children in accordance with its former type.

Finally, there is a `LOGOP`, or logic op. Like a `LISTOP`, this has one or more children, but it doesn't have an `op_last` field: so you have to follow `op_first` and then the `OpSIBLING` chain itself to find the last child. Instead it has an `op_other` field, which is comparable to the `op_next` field described below, and represents an alternate execution path. Operators like `and`, `or` and `?` are `LOGOP`s. Note that in general, `op_other` may not point to any of the direct children of the `LOGOP`.

Starting in version 5.21.2, perls built with the experimental define `-DPERL_OP_PARENT` add an extra boolean flag for each op, `op_moresib`. When not set, this indicates that this is the last op in an `OpSIBLING` chain. This frees up the `op_sibling` field on the last sibling to point back to the parent op. Under this build, that field is also renamed `op_sibparent` to reflect its joint role. The macro `OpSIBLING(o)` wraps this special behaviour, and always returns NULL on the last sibling. With this build the `op_parent(o)` function can be used to find the parent of any op. Thus for forward compatibility, you should always use the `OpSIBLING(o)` macro rather than accessing `op_sibling` directly.

Another way to examine the tree is to use a compiler back-end module, such as *B::Concise*.

Compile pass 1: check routines

The tree is created by the 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). Optimizations performed at this stage are subject to the same restrictions as in the pass 2.

Peephole optimizations are done by calling the function pointed to by the global variable `PL_peepp`. By default, `PL_peepp` just calls the function pointed to by the global variable `PL_rpeepp`. By default, that performs some basic op fixups and optimisations along the execution-order op chain, and recursively calls `PL_rpeepp` for each side chain of ops (resulting from conditionals). Extensions may provide additional optimisations or fixups, hooking into either the per-subroutine or recursive stage, like this:

```
static peep_t prev_peepp;
static void my_peep(pTHX_ OP *o)
{
    /* custom per-subroutine optimisation goes here */
    prev_peepp(aTHX_ o);
    /* custom per-subroutine optimisation may also go here */
}
BOOT:
```

```
        prev_peepp = PL_peepp;
        PL_peepp = my_peep;

    static peep_t prev_rpeepp;
    static void my_rpeep(pTHX_ OP *o)
    {
        OP *orig_o = o;
        for(; o; o = o->op_next) {
            /* custom per-op optimisation goes here */
        }
        prev_rpeepp(aTHX_ orig_o);
    }
    BOOT:
        prev_rpeepp = PL_rpeepp;
        PL_rpeepp = my_rpeep;
```

Pluggable runops

The compile tree is executed in a runops function. There are two runops functions, in *run.c* and in *dump.c*. `Perl_runops_debug` is used with DEBUGGING and `Perl_runops_standard` is used otherwise. For fine control over the execution of the compile tree it is possible to provide your own runops function.

It's probably best to copy one of the existing runops functions and change it to suit your needs. Then, in the BOOT section of your XS file, add the line:

```
  PL_runops = my_runops;
```

This function should be as efficient as possible to keep your programs running as fast as possible.

Compile-time scope hooks

As of perl 5.14 it is possible to hook into the compile-time lexical scope mechanism using `Perl_blockhook_register`. This is used like this:

```
    STATIC void my_start_hook(pTHX_ int full);
    STATIC BHK my_hooks;

    BOOT:
        BhkENTRY_set(&my_hooks, bhk_start, my_start_hook);
        Perl_blockhook_register(aTHX_ &my_hooks);
```

This will arrange to have `my_start_hook` called at the start of compiling every lexical scope. The available hooks are:

`void bhk_start(pTHX_ int full)`

This is called just after starting a new lexical scope. Note that Perl code like

```
    if ($x) { ... }
```

creates two scopes: the first starts at the `(` and has `full == 1`, the second starts at the `{` and has `full == 0`. Both end at the `}`, so calls to `start` and `pre`/`post_end` will match. Anything pushed onto the save stack by this hook will be popped just before the scope ends (between the `pre_` and `post_end` hooks, in fact).

`void bhk_pre_end(pTHX_ OP **o)`

This is called at the end of a lexical scope, just before unwinding the stack. *o* is the root of the optree representing the scope; it is a double pointer so you can replace the OP if you need to.

`void bhk_post_end(pTHX_ OP **o)`

This is called at the end of a lexical scope, just after unwinding the stack. *o* is as above. Note that it is possible for calls to `pre_` and `post_end` to nest, if there is something on the save stack that calls string eval.

`void bhk_eval(pTHX_ OP *const o)`

This is called just before starting to compile an `eval STRING`, `do FILE`, `require` or `use`, after the eval has been set up. *o* is the OP that requested the eval, and will normally be an `OP_ENTEREVAL`, `OP_DOFILE` or `OP_REQUIRE`.

Once you have your hook functions, you need a `BHK` structure to put them in. It's best to allocate it statically, since there is no way to free it once it's registered. The function pointers should be inserted into this structure using the `BhkENTRY_set` macro, which will also set flags indicating which entries are valid. If you do need to allocate your `BHK` dynamically for some reason, be sure to zero it before you start.

Once registered, there is no mechanism to switch these hooks off, so if that is necessary you will need to do this yourself. An entry in `%^H` is probably the best way, so the effect is lexically scoped; however it is also possible to use the `BhkDISABLE` and `BhkENABLE` macros to temporarily switch entries on and off. You should also be aware that generally speaking at least one scope will have opened before your extension is loaded, so you will see some `pre`/`post_end` pairs that didn't have a matching `start`.

Examining internal data structures with the dump functions

To aid debugging, the source file *dump.c* contains a number of functions which produce formatted output of internal data structures.

The most commonly used of these functions is `Perl_sv_dump`; it's used for dumping SVs, AVs, HVs, and CVs. The `Devel::Peek` module calls `sv_dump` to produce debugging output from Perl-space, so users of that module should already be familiar with its format.

`Perl_op_dump` can be used to dump an `OP` structure or any of its derivatives, and produces output similar to `perl -Dx`; in fact, `Perl_dump_eval` will dump the main root of the code being evaluated, exactly like `-Dx`.

Other useful functions are `Perl_dump_sub`, which turns a `GV` into an op tree, `Perl_dump_packsubs` which calls `Perl_dump_sub` on all the subroutines in a package like so: (Thankfully, these are all xsubs, so there is no op tree)

```
(gdb) print Perl_dump_packsubs(PL_defstash)

SUB attributes::bootstrap = (xsub 0x811fedc 0)

SUB UNIVERSAL::can = (xsub 0x811f50c 0)

SUB UNIVERSAL::isa = (xsub 0x811f304 0)

SUB UNIVERSAL::VERSION = (xsub 0x811f7ac 0)

SUB DynaLoader::boot_DynaLoader = (xsub 0x805b188 0)
```

and `Perl_dump_all`, which dumps all the subroutines in the stash and the op tree of the main root.

How multiple interpreters and concurrency are supported

Background and PERL_IMPLICIT_CONTEXT

The Perl interpreter can be regarded as a closed box: it has an API for feeding it code or otherwise making it do things, but it also has functions for its own use. This smells a lot like an object, and there are ways for you to build Perl so that you can have multiple interpreters, with one interpreter represented either as a C structure, or inside a thread-specific structure. These structures contain all the context, the state of that interpreter.

One macro controls the major Perl build flavor: MULTIPLICITY. The MULTIPLICITY build has a C structure that packages all the interpreter state. With multiplicity-enabled perls, PERL_IMPLICIT_CONTEXT is also normally defined, and enables the support for passing in a "hidden" first argument that represents all three data structures. MULTIPLICITY makes multi-threaded perls possible (with the ithreads threading model, related to the macro USE_ITHREADS.)

Two other "encapsulation" macros are the PERL_GLOBAL_STRUCT and PERL_GLOBAL_STRUCT_PRIVATE (the latter turns on the former, and the former turns on MULTIPLICITY.) The PERL_GLOBAL_STRUCT causes all the internal variables of Perl to be wrapped inside a single global struct, struct perl_vars, accessible as (globals) &PL_Vars or PL_VarsPtr or the function Perl_GetVars(). The PERL_GLOBAL_STRUCT_PRIVATE goes one step further, there is still a single struct (allocated in main() either from heap or from stack) but there are no global data symbols pointing to it. In either case the global struct should be initialized as the very first thing in main() using Perl_init_global_struct() and correspondingly tear it down after perl_free() using Perl_free_global_struct(), please see *miniperlmain.c* for usage details. You may also need to use dVAR in your coding to "declare the global variables" when you are using them. dTHX does this for you automatically.

To see whether you have non-const data you can use a BSD (or GNU) compatible `nm`:

```
nm libperl.a | grep -v ' [TURtr] '
```

If this displays any `D` or `d` symbols (or possibly `C` or `c`), you have non-const data. The symbols the `grep` removed are as follows: `Tt` are *text*, or code, the `Rr` are *read-only* (const) data, and the `U` is <undefined>, external symbols referred to.

The test *t/porting/libperl.t* does this kind of symbol sanity checking on `libperl.a`.

For backward compatibility reasons defining just PERL_GLOBAL_STRUCT doesn't actually hide all symbols inside a big global struct: some PerlIO_xxx vtables are left visible. The PERL_GLOBAL_STRUCT_PRIVATE then hides everything (see how the PERLIO_FUNCS_DECL is used).

All this obviously requires a way for the Perl internal functions to be either subroutines taking some kind of structure as the first argument, or subroutines taking nothing as the first argument. To enable these two very different ways of building the interpreter, the Perl source (as it does in so many other situations) makes heavy use of macros and subroutine naming conventions.

First problem: deciding which functions will be public API functions and which will be private. All functions whose names begin `S_` are private (think "S" for "secret" or "static"). All other functions begin with "Perl_", but just because a function begins with "Perl_" does not mean it is part of the API. (See *Internal Functions*.) The easiest way to be **sure** a function is part of the API is to find its entry in *perlapi*. If it exists in *perlapi*, it's part of the API. If it doesn't, and you think it should be (i.e., you need it for your extension), send mail via *perlbug* explaining why you think it should be.

Second problem: there must be a syntax so that the same subroutine declarations and calls can pass a structure as their first argument, or pass nothing. To solve this, the subroutines are named and declared in a particular way. Here's a typical start of a static function used within the Perl guts:

```
STATIC void
S_incline(pTHX_ char *s)
```

STATIC becomes "static" in C, and may be #define'd to nothing in some configurations in the future.

A public function (i.e. part of the internal API, but not necessarily sanctioned for use in extensions) begins like this:

```
void
Perl_sv_setiv(pTHX_ SV* dsv, IV num)
```

pTHX_ is one of a number of macros (in *perl.h*) that hide the details of the interpreter's context. THX stands for "thread", "this", or "thingy", as the case may be. (And no, George Lucas is not involved. :-) The first character could be 'p' for a **p**rototype, 'a' for **a**rgument, or 'd' for **d**eclaration, so we have pTHX, aTHX and dTHX, and their variants.

When Perl is built without options that set PERL_IMPLICIT_CONTEXT, there is no first argument containing the interpreter's context. The trailing underscore in the pTHX_ macro indicates that the macro expansion needs a comma after the context argument because other arguments follow it. If PERL_IMPLICIT_CONTEXT is not defined, pTHX_ will be ignored, and the subroutine is not prototyped to take the extra argument. The form of the macro without the trailing underscore is used when there are no additional explicit arguments.

When a core function calls another, it must pass the context. This is normally hidden via macros. Consider sv_setiv. It expands into something like this:

```
#ifdef PERL_IMPLICIT_CONTEXT
  #define sv_setiv(a,b)      Perl_sv_setiv(aTHX_ a, b)
  /* can't do this for vararg functions, see below */
#else
  #define sv_setiv           Perl_sv_setiv
#endif
```

This works well, and means that XS authors can gleefully write:

```
sv_setiv(foo, bar);
```

and still have it work under all the modes Perl could have been compiled with.

This doesn't work so cleanly for varargs functions, though, as macros imply that the number of arguments is known in advance. Instead we either need to spell them out fully, passing aTHX_ as the first argument (the Perl core tends to do this with functions like Perl_warner), or use a context-free version.

The context-free version of Perl_warner is called Perl_warner_nocontext, and does not take the extra argument. Instead it does dTHX; to get the context from thread-local storage. We `#define warner Perl_warner_nocontext` so that extensions get source compatibility at the expense of performance. (Passing an arg is cheaper than grabbing it from thread-local storage.)

You can ignore [pad]THXx when browsing the Perl headers/sources. Those are strictly for use within the core. Extensions and embedders need only be aware of [pad]THX.

So what happened to dTHR?

dTHR was introduced in perl 5.005 to support the older thread model. The older thread model now uses the THX mechanism to pass context pointers around, so dTHR is not useful any more. Perl 5.6.0 and later still have it for backward source compatibility, but it is defined to be a no-op.

How do I use all this in extensions?

When Perl is built with PERL_IMPLICIT_CONTEXT, extensions that call any functions in the Perl API will need to pass the initial context argument somehow. The kicker is that you will need to write it in such a way that the extension still compiles when Perl hasn't been built with

PERL_IMPLICIT_CONTEXT enabled.

There are three ways to do this. First, the easy but inefficient way, which is also the default, in order to maintain source compatibility with extensions: whenever *XSUB.h* is #included, it redefines the aTHX and aTHX_ macros to call a function that will return the context. Thus, something like:

```
sv_setiv(sv, num);
```

in your extension will translate to this when PERL_IMPLICIT_CONTEXT is in effect:

```
Perl_sv_setiv(Perl_get_context(), sv, num);
```

or to this otherwise:

```
Perl_sv_setiv(sv, num);
```

You don't have to do anything new in your extension to get this; since the Perl library provides Perl_get_context(), it will all just work.

The second, more efficient way is to use the following template for your Foo.xs:

```
#define PERL_NO_GET_CONTEXT     /* we want efficiency */
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

STATIC void my_private_function(int arg1, int arg2);

STATIC void
my_private_function(int arg1, int arg2)
{
    dTHX;       /* fetch context */
    ... call many Perl API functions ...
}

[... etc ...]

MODULE = Foo            PACKAGE = Foo

/* typical XSUB */

void
my_xsub(arg)
        int arg
    CODE:
        my_private_function(arg, 10);
```

Note that the only two changes from the normal way of writing an extension is the addition of a `#define PERL_NO_GET_CONTEXT` before including the Perl headers, followed by a `dTHX;` declaration at the start of every function that will call the Perl API. (You'll know which functions need this, because the C compiler will complain that there's an undeclared identifier in those functions.) No changes are needed for the XSUBs themselves, because the XS() macro is correctly defined to pass in the implicit context if needed.

The third, even more efficient way is to ape how it is done within the Perl guts:

```
#define PERL_NO_GET_CONTEXT     /* we want efficiency */
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

/* pTHX_ only needed for functions that call Perl API */
STATIC void my_private_function(pTHX_ int arg1, int arg2);

STATIC void
my_private_function(pTHX_ int arg1, int arg2)
{
    /* dTHX; not needed here, because THX is an argument */
    ... call Perl API functions ...
}

[... etc ...]

MODULE = Foo            PACKAGE = Foo

/* typical XSUB */

void
my_xsub(arg)
        int arg
    CODE:
        my_private_function(aTHX_ arg, 10);
```

This implementation never has to fetch the context using a function call, since it is always passed as an extra argument. Depending on your needs for simplicity or efficiency, you may mix the previous two approaches freely.

Never add a comma after pTHX yourself--always use the form of the macro with the underscore for functions that take explicit arguments, or the form without the argument for functions with no explicit arguments.

If one is compiling Perl with the `-DPERL_GLOBAL_STRUCT` the dVAR definition is needed if the Perl global variables (see *perlvars.h* or *globvar.sym*) are accessed in the function and dTHX is not used (the dTHX includes the dVAR if necessary). One notices the need for dVAR only with the said compile-time define, because otherwise the Perl global variables are visible as-is.

Should I do anything special if I call perl from multiple threads?

If you create interpreters in one thread and then proceed to call them in another, you need to make sure perl's own Thread Local Storage (TLS) slot is initialized correctly in each of those threads.

The `perl_alloc` and `perl_clone` API functions will automatically set the TLS slot to the interpreter they created, so that there is no need to do anything special if the interpreter is always accessed in the same thread that created it, and that thread did not create or call any other interpreters afterwards. If that is not the case, you have to set the TLS slot of the thread before calling any functions in the Perl API on that particular interpreter. This is done by calling the PERL_SET_CONTEXT macro in that thread as the first thing you do:

```
/* do this before doing anything else with some_perl */
PERL_SET_CONTEXT(some_perl);

... other Perl API calls on some_perl go here ...
```

Future Plans and PERL_IMPLICIT_SYS

Just as PERL_IMPLICIT_CONTEXT provides a way to bundle up everything that the interpreter knows about itself and pass it around, so too are there plans to allow the interpreter to bundle up everything it knows about the environment it's running on. This is enabled with the PERL_IMPLICIT_SYS macro. Currently it only works with USE_ITHREADS on Windows.

This allows the ability to provide an extra pointer (called the "host" environment) for all the system calls. This makes it possible for all the system stuff to maintain their own state, broken down into seven C structures. These are thin wrappers around the usual system calls (see *win32/perllib.c*) for the default perl executable, but for a more ambitious host (like the one that would do fork() emulation) all the extra work needed to pretend that different interpreters are actually different "processes", would be done here.

The Perl engine/interpreter and the host are orthogonal entities. There could be one or more interpreters in a process, and one or more "hosts", with free association between them.

Internal Functions

All of Perl's internal functions which will be exposed to the outside world are prefixed by `Perl_` so that they will not conflict with XS functions or functions used in a program in which Perl is embedded. Similarly, all global variables begin with `PL_`. (By convention, static functions start with `S_`.)

Inside the Perl core (`PERL_CORE` defined), you can get at the functions either with or without the `Perl_` prefix, thanks to a bunch of defines that live in *embed.h*. Note that extension code should *not* set `PERL_CORE`; this exposes the full perl internals, and is likely to cause breakage of the XS in each new perl release.

The file *embed.h* is generated automatically from *embed.pl* and *embed.fnc*. *embed.pl* also creates the prototyping header files for the internal functions, generates the documentation and a lot of other bits and pieces. It's important that when you add a new function to the core or change an existing one, you change the data in the table in *embed.fnc* as well. Here's a sample entry from that table:

```
Apd |SV**   |av_fetch   |AV* ar|I32 key|I32 lval
```

The second column is the return type, the third column the name. Columns after that are the arguments. The first column is a set of flags:

A

This function is a part of the public API. All such functions should also have 'd', very few do not.

p

This function has a `Perl_` prefix; i.e. it is defined as `Perl_av_fetch`.

d

This function has documentation using the `apidoc` feature which we'll look at in a second. Some functions have 'd' but not 'A'; docs are good.

Other available flags are:

s

This is a static function and is defined as `STATIC S_whatever`, and usually called within the sources as `whatever(...)`.

n

This does not need an interpreter context, so the definition has no `pTHX`, and it follows that callers don't use `aTHX`. (See *Background and PERL_IMPLICIT_CONTEXT*.)

r

This function never returns; `croak`, `exit` and friends.

f

This function takes a variable number of arguments, `printf` style. The argument list should end with `...`, like this:

```
Afprd   |void   |croak          |const char* pat|...
```

M

This function is part of the experimental development API, and may change or disappear without notice.

o

This function should not have a compatibility macro to define, say, `Perl_parse` to `parse`. It must be called as `Perl_parse`.

x

This function isn't exported out of the Perl core.

m

This is implemented as a macro.

X

This function is explicitly exported.

E

This function is visible to extensions included in the Perl core.

b

Binary backward compatibility; this function is a macro but also has a `Perl_` implementation (which is exported).

others

See the comments at the top of `embed.fnc` for others.

If you edit *embed.pl* or *embed.fnc*, you will need to run `make regen_headers` to force a rebuild of *embed.h* and other auto-generated files.

Formatted Printing of IVs, UVs, and NVs

If you are printing IVs, UVs, or NVS instead of the stdio(3) style formatting codes like `%d`, `%ld`, `%f`, you should use the following macros for portability

```
IVdf            IV in decimal
UVuf            UV in decimal
UVof            UV in octal
UVxf            UV in hexadecimal
NVef            NV %e-like
NVff            NV %f-like
NVgf            NV %g-like
```

These will take care of 64-bit integers and long doubles. For example:

```
printf("IV is %"IVdf"\n", iv);
```

The IVdf will expand to whatever is the correct format for the IVs.

Note that there are different "long doubles": Perl will use whatever the compiler has.

If you are printing addresses of pointers, use UVxf combined with PTR2UV(), do not use %lx or %p.

Formatted Printing of Size_t and SSize_t

The most general way to do this is to cast them to a UV or IV, and print as in the *previous section*.

But if you're using `PerlIO_printf()`, it's less typing and visual clutter to use the `"%z"` length modifier (for *siZe*):

```
PerlIO_printf("STRLEN is %zu\n", len);
```

This modifier is not portable, so its use should be restricted to `PerlIO_printf()`.

Pointer-To-Integer and Integer-To-Pointer

Because pointer size does not necessarily equal integer size, use the follow macros to do it right.

```
PTR2UV(pointer)
PTR2IV(pointer)
PTR2NV(pointer)
INT2PTR(pointertotype, integer)
```

For example:

```
IV  iv = ...;
SV *sv = INT2PTR(SV*, iv);
```

and

```
AV *av = ...;
UV  uv = PTR2UV(av);
```

Exception Handling

There are a couple of macros to do very basic exception handling in XS modules. You have to define `NO_XSLOCKS` before including *XSUB.h* to be able to use these macros:

```
#define NO_XSLOCKS
#include "XSUB.h"
```

You can use these macros if you call code that may croak, but you need to do some cleanup before giving control back to Perl. For example:

```
dXCPT;    /* set up necessary variables */

XCPT_TRY_START {
  code_that_may_croak();
} XCPT_TRY_END

XCPT_CATCH
{
  /* do cleanup here */
  XCPT_RETHROW;
}
```

Note that you always have to rethrow an exception that has been caught. Using these macros, it is not possible to just catch the exception and ignore it. If you have to ignore the exception, you have to use the `call_*` function.

The advantage of using the above macros is that you don't have to setup an extra function for `call_*`, and that using these macros is faster than using `call_*`.

Source Documentation

There's an effort going on to document the internal functions and automatically produce reference manuals from them -- *perlapi* is one such manual which details all the functions which are available to XS writers. *perlintern* is the autogenerated manual for the functions which are not part of the API and are supposedly for internal use only.

Source documentation is created by putting POD comments into the C source, like this:

```
/*
=for apidoc sv_setiv

Copies an integer into the given SV.  Does not handle 'set' magic.  See
L<perlapi/sv_setiv_mg>.

=cut
*/
```

Please try and supply some documentation if you add functions to the Perl core.

Backwards compatibility

The Perl API changes over time. New functions are added or the interfaces of existing functions are changed. The `Devel::PPPort` module tries to provide compatibility code for some of these changes, so XS writers don't have to code it themselves when supporting multiple versions of Perl.

`Devel::PPPort` generates a C header file *ppport.h* that can also be run as a Perl script. To generate *ppport.h*, run:

```
    perl -MDevel::PPPort -eDevel::PPPort::WriteFile
```

Besides checking existing XS code, the script can also be used to retrieve compatibility information for various API calls using the `--api-info` command line switch. For example:

```
  % perl ppport.h --api-info=sv_magicext
```

For details, see `perldoc ppport.h`.

Unicode Support

Perl 5.6.0 introduced Unicode support. It's important for porters and XS writers to understand this support and make sure that the code they write does not corrupt Unicode data.

What is Unicode, anyway?

In the olden, less enlightened times, we all used to use ASCII. Most of us did, anyway. The big problem with ASCII is that it's American. Well, no, that's not actually the problem; the problem is that it's not particularly useful for people who don't use the Roman alphabet. What used to happen was that particular languages would stick their own alphabet in the upper range of the sequence, between 128 and 255. Of course, we then ended up with plenty of variants that weren't quite ASCII, and the whole point of it being a standard was lost.

Worse still, if you've got a language like Chinese or Japanese that has hundreds or thousands of characters, then you really can't fit them into a mere 256, so they had to forget about ASCII altogether, and build their own systems using pairs of numbers to refer to one character.

To fix this, some people formed Unicode, Inc. and produced a new character set containing all the characters you can possibly think of and more. There are several ways of representing these

characters, and the one Perl uses is called UTF-8. UTF-8 uses a variable number of bytes to represent a character. You can learn more about Unicode and Perl's Unicode model in *perlunicode*.

(On EBCDIC platforms, Perl uses instead UTF-EBCDIC, which is a form of UTF-8 adapted for EBCDIC platforms. Below, we just talk about UTF-8. UTF-EBCDIC is like UTF-8, but the details are different. The macros hide the differences from you, just remember that the particular numbers and bit patterns presented below will differ in UTF-EBCDIC.)

How can I recognise a UTF-8 string?

You can't. This is because UTF-8 data is stored in bytes just like non-UTF-8 data. The Unicode character 200, (`0xC8` for you hex types) capital E with a grave accent, is represented by the two bytes `v196.172`. Unfortunately, the non-Unicode string `chr(196).chr(172)` has that byte sequence as well. So you can't tell just by looking -- this is what makes Unicode input an interesting problem.

In general, you either have to know what you're dealing with, or you have to guess. The API function `is_utf8_string` can help; it'll tell you if a string contains only valid UTF-8 characters, and the chances of a non-UTF-8 string looking like valid UTF-8 become very small very quickly with increasing string length. On a character-by-character basis, `isUTF8_CHAR` will tell you whether the current character in a string is valid UTF-8.

How does UTF-8 represent Unicode characters?

As mentioned above, UTF-8 uses a variable number of bytes to store a character. Characters with values 0...127 are stored in one byte, just like good ol' ASCII. Character 128 is stored as `v194.128`; this continues up to character 191, which is `v194.191`. Now we've run out of bits (191 is binary `10111111`) so we move on; character 192 is `v195.128`. And so it goes on, moving to three bytes at character 2048. *"Unicode Encodings" in perlunicode* has pictures of how this works.

Assuming you know you're dealing with a UTF-8 string, you can find out how long the first character in it is with the `UTF8SKIP` macro:

```
char *utf = "\305\233\340\240\201";
I32 len;

len = UTF8SKIP(utf); /* len is 2 here */
utf += len;
len = UTF8SKIP(utf); /* len is 3 here */
```

Another way to skip over characters in a UTF-8 string is to use `utf8_hop`, which takes a string and a number of characters to skip over. You're on your own about bounds checking, though, so don't use it lightly.

All bytes in a multi-byte UTF-8 character will have the high bit set, so you can test if you need to do something special with this character like this (the `UTF8_IS_INVARIANT()` is a macro that tests whether the byte is encoded as a single byte even in UTF-8):

```
U8 *utf;
U8 *utf_end; /* 1 beyond buffer pointed to by utf */
UV uv; /* Note: a UV, not a U8, not a char */
STRLEN len; /* length of character in bytes */

if (!UTF8_IS_INVARIANT(*utf))
    /* Must treat this as UTF-8 */
    uv = utf8_to_uvchr_buf(utf, utf_end, &len);
else
    /* OK to treat this character as a byte */
    uv = *utf;
```

You can also see in that example that we use `utf8_to_uvchr_buf` to get the value of the character; the inverse function `uvchr_to_utf8` is available for putting a UV into UTF-8:

```
if (!UVCHR_IS_INVARIANT(uv))
    /* Must treat this as UTF8 */
    utf8 = uvchr_to_utf8(utf8, uv);
else
    /* OK to treat this character as a byte */
    *utf8++ = uv;
```

You **must** convert characters to UVs using the above functions if you're ever in a situation where you have to match UTF-8 and non-UTF-8 characters. You may not skip over UTF-8 characters in this case. If you do this, you'll lose the ability to match hi-bit non-UTF-8 characters; for instance, if your UTF-8 string contains `v196.172`, and you skip that character, you can never match a `chr(200)` in a non-UTF-8 string. So don't do that!

(Note that we don't have to test for invariant characters in the examples above. The functions work on any well-formed UTF-8 input. It's just that its faster to avoid the function overhead when it's not needed.)

How does Perl store UTF-8 strings?

Currently, Perl deals with UTF-8 strings and non-UTF-8 strings slightly differently. A flag in the SV, `SVf_UTF8`, indicates that the string is internally encoded as UTF-8. Without it, the byte value is the codepoint number and vice versa. This flag is only meaningful if the SV is `SvPOK` or immediately after stringification via `SvPV` or a similar macro. You can check and manipulate this flag with the following macros:

```
SvUTF8(sv)
SvUTF8_on(sv)
SvUTF8_off(sv)
```

This flag has an important effect on Perl's treatment of the string: if UTF-8 data is not properly distinguished, regular expressions, `length`, `substr` and other string handling operations will have undesirable (wrong) results.

The problem comes when you have, for instance, a string that isn't flagged as UTF-8, and contains a byte sequence that could be UTF-8 -- especially when combining non-UTF-8 and UTF-8 strings.

Never forget that the `SVf_UTF8` flag is separate from the PV value; you need to be sure you don't accidentally knock it off while you're manipulating SVs. More specifically, you cannot expect to do this:

```
SV *sv;
SV *nsv;
STRLEN len;
char *p;

p = SvPV(sv, len);
frobnicate(p);
nsv = newSVpvn(p, len);
```

The `char*` string does not tell you the whole story, and you can't copy or reconstruct an SV just by copying the string value. Check if the old SV has the UTF8 flag set (*after* the `SvPV` call), and act accordingly:

```
p = SvPV(sv, len);
is_utf8 = SvUTF8(sv);
frobnicate(p, is_utf8);
```

```
nsv = newSVpvn(p, len);
if (is_utf8)
    SvUTF8_on(nsv);
```

In the above, your `frobnicate` function has been changed to be made aware of whether or not it's dealing with UTF-8 data, so that it can handle the string appropriately.

Since just passing an SV to an XS function and copying the data of the SV is not enough to copy the UTF8 flags, even less right is just passing a `char *` to an XS function.

For full generality, use the *`DO_UTF8`* macro to see if the string in an SV is to be *treated* as UTF-8. This takes into account if the call to the XS function is being made from within the scope of `use bytes`. If so, the underlying bytes that comprise the UTF-8 string are to be exposed, rather than the character they represent. But this pragma should only really be used for debugging and perhaps low-level testing at the byte level. Hence most XS code need not concern itself with this, but various areas of the perl core do need to support it.

And this isn't the whole story. Starting in Perl v5.12, strings that aren't encoded in UTF-8 may also be treated as Unicode under various conditions (see *"ASCII Rules versus Unicode Rules" in perlunicode*). This is only really a problem for characters whose ordinals are between 128 and 255, and their behavior varies under ASCII versus Unicode rules in ways that your code cares about (see *"The "Unicode Bug"" in perlunicode*). There is no published API for dealing with this, as it is subject to change, but you can look at the code for `pp_lc` in *pp.c* for an example as to how it's currently done.

How do I convert a string to UTF-8?

If you're mixing UTF-8 and non-UTF-8 strings, it is necessary to upgrade the non-UTF-8 strings to UTF-8. If you've got an SV, the easiest way to do this is:

```
sv_utf8_upgrade(sv);
```

However, you must not do this, for example:

```
if (!SvUTF8(left))
    sv_utf8_upgrade(left);
```

If you do this in a binary operator, you will actually change one of the strings that came into the operator, and, while it shouldn't be noticeable by the end user, it can cause problems in deficient code.

Instead, `bytes_to_utf8` will give you a UTF-8-encoded **copy** of its string argument. This is useful for having the data available for comparisons and so on, without harming the original SV. There's also `utf8_to_bytes` to go the other way, but naturally, this will fail if the string contains any characters above 255 that can't be represented in a single byte.

How do I compare strings?

"sv_cmp" in perlapi and *"sv_cmp_flags" in perlapi* do a lexigraphic comparison of two SV's, and handle UTF-8ness properly. Note, however, that Unicode specifies a much fancier mechanism for collation, available via the *Unicode::Collate* module.

To just compare two strings for equality/non-equality, you can just use *`memEQ()`* and *`memNE()`* as usual, except the strings must be both UTF-8 or not UTF-8 encoded.

To compare two strings case-insensitively, use *`foldEQ_utf8()`* (the strings don't have to have the same UTF-8ness).

Is there anything else I need to know?

Not really. Just remember these things:

- There's no way to tell if a `char *` or `U8 *` string is UTF-8 or not. But you can tell if an SV is to be treated as UTF-8 by calling `DO_UTF8` on it, after stringifying it with `SvPV` or a similar macro. And, you can tell if SV is actually UTF-8 (even if it is not to be treated as such) by looking at its `SvUTF8` flag (again after stringifying it). Don't forget to set the flag if something should be UTF-8. Treat the flag as part of the PV, even though it's not -- if you pass on the PV to somewhere, pass on the flag too.

- If a string is UTF-8, **always** use `utf8_to_uvchr_buf` to get at the value, unless `UTF8_IS_INVARIANT(*s)` in which case you can use `*s`.

- When writing a character UV to a UTF-8 string, **always** use `uvchr_to_utf8`, unless `UVCHR_IS_INVARIANT(uv))` in which case you can use `*s = uv`.

- Mixing UTF-8 and non-UTF-8 strings is tricky. Use `bytes_to_utf8` to get a new string which is UTF-8 encoded, and then combine them.

Custom Operators

Custom operator support is an experimental feature that allows you to define your own ops. This is primarily to allow the building of interpreters for other languages in the Perl core, but it also allows optimizations through the creation of "macro-ops" (ops which perform the functions of multiple ops which are usually executed together, such as `gvsv, gvsv, add`.)

This feature is implemented as a new op type, `OP_CUSTOM`. The Perl core does not "know" anything special about this op type, and so it will not be involved in any optimizations. This also means that you can define your custom ops to be any op structure -- unary, binary, list and so on -- you like.

It's important to know what custom operators won't do for you. They won't let you add new syntax to Perl, directly. They won't even let you add new keywords, directly. In fact, they won't change the way Perl compiles a program at all. You have to do those changes yourself, after Perl has compiled the program. You do this either by manipulating the op tree using a `CHECK` block and the `B::Generate` module, or by adding a custom peephole optimizer with the `optimize` module.

When you do this, you replace ordinary Perl ops with custom ops by creating ops with the type `OP_CUSTOM` and the `op_ppaddr` of your own PP function. This should be defined in XS code, and should look like the PP ops in `pp_*.c`. You are responsible for ensuring that your op takes the appropriate number of values from the stack, and you are responsible for adding stack marks if necessary.

You should also "register" your op with the Perl interpreter so that it can produce sensible error and warning messages. Since it is possible to have multiple custom ops within the one "logical" op type `OP_CUSTOM`, Perl uses the value of `o->op_ppaddr` to determine which custom op it is dealing with. You should create an `XOP` structure for each ppaddr you use, set the properties of the custom op with `XopENTRY_set`, and register the structure against the ppaddr using `Perl_custom_op_register`. A trivial example might look like:

```
static XOP my_xop;
static OP *my_pp(pTHX);

BOOT:
    XopENTRY_set(&my_xop, xop_name, "myxop");
    XopENTRY_set(&my_xop, xop_desc, "Useless custom op");
    Perl_custom_op_register(aTHX_ my_pp, &my_xop);
```

The available fields in the structure are:

xop_name

A short name for your op. This will be included in some error messages, and will also be returned as `$op->name` by the *B* module, so it will appear in the output of module like

B::Concise.

xop_desc

A short description of the function of the op.

xop_class

Which of the various `*OP` structures this op uses. This should be one of the `OA_*` constants from *op.h*, namely

OA_BASEOP

OA_UNOP

OA_BINOP

OA_LOGOP

OA_LISTOP

OA_PMOP

OA_SVOP

OA_PADOP

OA_PVOP_OR_SVOP

This should be interpreted as 'PVOP' only. The `_OR_SVOP` is because the only core `PVOP`, `OP_TRANS`, can sometimes be a `SVOP` instead.

OA_LOOP

OA_COP

The other `OA_*` constants should not be used.

xop_peep

This member is of type `Perl_cpeep_t`, which expands to `void (*Perl_cpeep_t)(aTHX_ OP *o, OP *oldop)`. If it is set, this function will be called from `Perl_rpeep` when ops of this type are encountered by the peephole optimizer. *o* is the OP that needs optimizing; *oldop* is the previous OP optimized, whose `op_next` points to *o*.

`B::Generate` directly supports the creation of custom ops by name.

Dynamic Scope and the Context Stack

Note: this section describes a non-public internal API that is subject to change without notice.

Introduction to the context stack

In Perl, dynamic scoping refers to the runtime nesting of things like subroutine calls, evals etc, as well as the entering and exiting of block scopes. For example, the restoring of a `local`ised variable is determined by the dynamic scope.

Perl tracks the dynamic scope by a data structure called the context stack, which is an array of `PERL_CONTEXT` structures, and which is itself a big union for all the types of context. Whenever a new scope is entered (such as a block, a `for` loop, or a subroutine call), a new context entry is pushed onto the stack. Similarly when leaving a block or returning from a subroutine call etc. a context is popped. Since the context stack represents the current dynamic scope, it can be searched. For example, `next LABEL` searches back through the stack looking for a loop context that matches the label; `return` pops contexts until it finds a sub or eval context or similar; `caller` examines sub contexts on the stack.

Each context entry is labelled with a context type, `cx_type`. Typical context types are `CXt_SUB`, `CXt_EVAL` etc., as well as `CXt_BLOCK` and `CXt_NULL` which represent a basic scope (as pushed by `pp_enter`) and a sort block. The type determines which part of the context union are valid.

The main division in the context struct is between a substitution scope (CXt_SUBST) and block scopes, which are everything else. The former is just used while executing s///e, and won't be discussed further here.

All the block scope types share a common base, which corresponds to CXt_BLOCK. This stores the old values of various scope-related variables like PL_curpm, as well as information about the current scope, such as gimme. On scope exit, the old variables are restored.

Particular block scope types store extra per-type information. For example, CXt_SUB stores the currently executing CV, while the various for loop types might hold the original loop variable SV. On scope exit, the per-type data is processed; for example the CV has its reference count decremented, and the original loop variable is restored.

The macro cxstack returns the base of the current context stack, while cxstack_ix is the index of the current frame within that stack.

In fact, the context stack is actually part of a stack-of-stacks system; whenever something unusual is done such as calling a DESTROY or tie handler, a new stack is pushed, then popped at the end.

Note that the API described here changed considerably in perl 5.24; prior to that, big macros like PUSHBLOCK and POPSUB were used; in 5.24 they were replaced by the inline static functions described below. In addition, the ordering and detail of how these macros/function work changed in many ways, often subtly. In particular they didn't handle saving the savestack and temps stack positions, and required additional ENTER, SAVETMPS and LEAVE compared to the new functions. The old-style macros will not be described further.

Pushing contexts

For pushing a new context, the two basic functions are cx = cx_pushblock(), which pushes a new basic context block and returns its address, and a family of similar functions with names like cx_pushsub(cx) which populate the additional type-dependent fields in the cx struct. Note that CXt_NULL and CXt_BLOCK don't have their own push functions, as they don't store any data beyond that pushed by cx_pushblock.

The fields of the context struct and the arguments to the cx_* functions are subject to change between perl releases, representing whatever is convenient or efficient for that release.

A typical context stack pushing can be found in pp_entersub; the following shows a simplified and stripped-down example of a non-XS call, along with comments showing roughly what each function does.

```
dMARK;
U8 gimme      = GIMME_V;
bool hasargs  = cBOOL(PL_op->op_flags & OPf_STACKED);
OP *retop     = PL_op->op_next;
I32 old_ss_ix = PL_savestack_ix;
CV *cv        = ....;

/* ... make mortal copies of stack args which are PADTMPs here ... */

/* ... do any additional savestack pushes here ... */

/* Now push a new context entry of type 'CXt_SUB'; initially just
 * doing the actions common to all block types: */

cx = cx_pushblock(CXt_SUB, gimme, MARK, old_ss_ix);

    /* this does (approximately):
```

```
        CXINC;              /* cxstack_ix++ (grow if necessary) */
        cx = CX_CUR();      /* and get the address of new frame */
        cx->cx_type        = CXt_SUB;
        cx->blk_gimme      = gimme;
        cx->blk_oldsp      = MARK - PL_stack_base;
        cx->blk_oldsaveix  = old_ss_ix;
        cx->blk_oldcop     = PL_curcop;
        cx->blk_oldmarksp  = PL_markstack_ptr - PL_markstack;
        cx->blk_oldscopesp = PL_scopestack_ix;
        cx->blk_oldpm      = PL_curpm;
        cx->blk_old_tmpsfloor = PL_tmps_floor;

        PL_tmps_floor        = PL_tmps_ix;
    */

/* then update the new context frame with subroutine-specific info,
 * such as the CV about to be executed: */

cx_pushsub(cx, cv, retop, hasargs);

    /* this does (approximately):
        cx->blk_sub.cv          = cv;
        cx->blk_sub.olddepth    = CvDEPTH(cv);
        cx->blk_sub.prevcomppad = PL_comppad;
        cx->cx_type            |= (hasargs) ? CXp_HASARGS : 0;
        cx->blk_sub.retop       = retop;
        SvREFCNT_inc_simple_void_NN(cv);
    */
```

Note that `cx_pushblock()` sets two new floors: for the args stack (to `MARK`) and the temps stack (to `PL_tmps_ix`). While executing at this scope level, every `nextstate` (amongst others) will reset the args and tmps stack levels to these floors. Note that since `cx_pushblock` uses the current value of `PL_tmps_ix` rather than it being passed as an arg, this dictates at what point `cx_pushblock` should be called. In particular, any new mortals which should be freed only on scope exit (rather than at the next `nextstate`) should be created first.

Most callers of `cx_pushblock` simply set the new args stack floor to the top of the previous stack frame, but for `CXt_LOOP_LIST` it stores the items being iterated over on the stack, and so sets `blk_oldsp` to the top of these items instead. Note that, contrary to its name, `blk_oldsp` doesn't always represent the value to restore `PL_stack_sp` to on scope exit.

Note the early capture of `PL_savestack_ix` to `old_ss_ix`, which is later passed as an arg to `cx_pushblock`. In the case of `pp_entersub`, this is because, although most values needing saving are stored in fields of the context struct, an extra value needs saving only when the debugger is running, and it doesn't make sense to bloat the struct for this rare case. So instead it is saved on the savestack. Since this value gets calculated and saved before the context is pushed, it is necessary to pass the old value of `PL_savestack_ix` to `cx_pushblock`, to ensure that the saved value gets freed during scope exit. For most users of `cx_pushblock`, where nothing needs pushing on the save stack, `PL_savestack_ix` is just passed directly as an arg to `cx_pushblock`.

Note that where possible, values should be saved in the context struct rather than on the save stack; it's much faster that way.

Normally `cx_pushblock` should be immediately followed by the appropriate `cx_pushfoo`, with nothing between them; this is because if code in-between could die (e.g. a warning upgraded to fatal), then the context stack unwinding code in `dounwind` would see (in the example above) a `CXt_SUB`

context frame, but without all the subroutine-specific fields set, and crashes would soon ensue.

Where the two must be separate, initially set the type to `CXt_NULL` or `CXt_BLOCK`, and later change it to `CXt_foo` when doing the `cx_pushfoo`. This is exactly what `pp_enteriter` does, once it's determined which type of loop it's pushing.

Popping contexts

Contexts are popped using `cx_popsub()` etc. and `cx_popblock()`. Note however, that unlike `cx_pushblock`, neither of these functions actually decrement the current context stack index; this is done separately using `CX_POP()`.

There are two main ways that contexts are popped. During normal execution as scopes are exited, functions like `pp_leave`, `pp_leaveloop` and `pp_leavesub` process and pop just one context using `cx_popfoo` and `cx_popblock`. On the other hand, things like `pp_return` and `next` may have to pop back several scopes until a sub or loop context is found, and exceptions (such as `die`) need to pop back contexts until an eval context is found. Both of these are accomplished by `dounwind()`, which is capable of processing and popping all contexts above the target one.

Here is a typical example of context popping, as found in `pp_leavesub` (simplified slightly):

```
U8 gimme;
PERL_CONTEXT *cx;
SV **oldsp;
OP *retop;

cx = CX_CUR();

gimme = cx->blk_gimme;
oldsp = PL_stack_base + cx->blk_oldsp; /* last arg of previous frame */

if (gimme == G_VOID)
    PL_stack_sp = oldsp;
else
    leave_adjust_stacks(oldsp, oldsp, gimme, 0);

CX_LEAVE_SCOPE(cx);
cx_popsub(cx);
cx_popblock(cx);
retop = cx->blk_sub.retop;
CX_POP(cx);

return retop;
```

The steps above are in a very specific order, designed to be the reverse order of when the context was pushed. The first thing to do is to copy and/or protect any any return arguments and free any temps in the current scope. Scope exits like an rvalue sub normally return a mortal copy of their return args (as opposed to lvalue subs). It is important to make this copy before the save stack is popped or variables are restored, or bad things like the following can happen:

```
sub f { my $x =...; $x }  # $x freed before we get to copy it
sub f { /(...)/;    $1 }  # PL_curpm restored before $1 copied
```

Although we wish to free any temps at the same time, we have to be careful not to free any temps which are keeping return args alive; nor to free the temps we have just created while mortal copying return args. Fortunately, `leave_adjust_stacks()` is capable of making mortal copies of return

args, shifting args down the stack, and only processing those entries on the temps stack that are safe to do so.

In void context no args are returned, so it's more efficient to skip calling `leave_adjust_stacks()`. Also in void context, a `nextstate` op is likely to be imminently called which will do a FREETMPS, so there's no need to do that either.

The next step is to pop savestack entries: `CX_LEAVE_SCOPE(cx)` is just defined as `<LEAVE_SCOPE(cx-blk_oldsaveix)>>`. Note that during the popping, it's possible for perl to call destructors, call STORE to undo localisations of tied vars, and so on. Any of these can die or call `exit()`. In this case, `dounwind()` will be called, and the current context stack frame will be re-processed. Thus it is vital that all steps in popping a context are done in such a way to support reentrancy. The other alternative, of decrementing `cxstack_ix` *before* processing the frame, would lead to leaks and the like if something died halfway through, or overwriting of the current frame.

CX_LEAVE_SCOPE itself is safely re-entrant: if only half the savestack items have been popped before dying and getting trapped by eval, then the CX_LEAVE_SCOPEs in `dounwind` or `pp_leaveeval` will continue where the first one left off.

The next step is the type-specific context processing; in this case `cx_popsub`. In part, this looks like:

```
cv = cx->blk_sub.cv;
CvDEPTH(cv) = cx->blk_sub.olddepth;
cx->blk_sub.cv = NULL;
SvREFCNT_dec(cv);
```

where its processing the just-executed CV. Note that before it decrements the CV's reference count, it nulls the `blk_sub.cv`. This means that if it re-enters, the CV won't be freed twice. It also means that you can't rely on such type-specific fields having useful values after the return from `cx_popfoo`.

Next, `cx_popblock` restores all the various interpreter vars to their previous values or previous high water marks; it expands to:

```
PL_markstack_ptr = PL_markstack + cx->blk_oldmarksp;
PL_scopestack_ix = cx->blk_oldscopesp;
PL_curpm         = cx->blk_oldpm;
PL_curcop        = cx->blk_oldcop;
PL_tmps_floor    = cx->blk_old_tmpsfloor;
```

Note that it *doesn't* restore `PL_stack_sp`; as mentioned earlier, which value to restore it to depends on the context type (specifically `for (list) {}`), and what args (if any) it returns; and that will already have been sorted out earlier by `leave_adjust_stacks()`.

Finally, the context stack pointer is actually decremented by `CX_POP(cx)`. After this point, it's possible that that the current context frame could be overwritten by other contexts being pushed. Although things like ties and DESTROY are supposed to work within a new context stack, it's best not to assume this. Indeed on debugging builds, `CX_POP(cx)` deliberately sets `cx` to null to detect code that is still relying on the field values in that context frame. Note in the `pp_leavesub()` example above, we grab `blk_sub.retop` *before* calling CX_POP.

Redoing contexts

Finally, there is `cx_topblock(cx)`, which acts like a super-`nextstate` as regards to resetting various vars to their base values. It is used in places like `pp_next`, `pp_redo` and `pp_goto` where rather than exiting a scope, we want to re-initialise the scope. As well as resetting `PL_stack_sp` like `nextstate`, it also resets `PL_markstack_ptr`, `PL_scopestack_ix` and `PL_curpm`. Note that it doesn't do a FREETMPS.

AUTHORS

Until May 1997, this document was maintained by Jeff Okamoto <okamoto@corp.hp.com>. It is now maintained as part of Perl itself by the Perl 5 Porters <perl5-porters@perl.org>.

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.

SEE ALSO

perlapi, *perlintern*, *perlxs*, *perlembed*