NAME

perlxs - XS language reference manual

DESCRIPTION

Introduction

XS is an interface description file format used to create an extension interface between Perl and C code (or a C library) which one wishes to use with Perl. The XS interface is combined with the library to create a new library which can then be either dynamically loaded or statically linked into perl. The XS interface description is written in the XS language and is the core component of the Perl extension interface.

An **XSUB** forms the basic unit of the XS interface. After compilation by the **xsubpp** compiler, each XSUB amounts to a C function definition which will provide the glue between Perl calling conventions and C calling conventions.

The glue code pulls the arguments from the Perl stack, converts these Perl values to the formats expected by a C function, call this C function, transfers the return values of the C function back to Perl. Return values here may be a conventional C return value or any C function arguments that may serve as output parameters. These return values may be passed back to Perl either by putting them on the Perl stack, or by modifying the arguments supplied from the Perl side.

The above is a somewhat simplified view of what really happens. Since Perl allows more flexible calling conventions than C, XSUBs may do much more in practice, such as checking input parameters for validity, throwing exceptions (or returning undef/empty list) if the return value from the C function indicates failure, calling different C functions based on numbers and types of the arguments, providing an object-oriented interface, etc.

Of course, one could write such glue code directly in C. However, this would be a tedious task, especially if one needs to write glue for multiple C functions, and/or one is not familiar enough with the Perl stack discipline and other such arcana. XS comes to the rescue here: instead of writing this glue C code in long-hand, one can write a more concise short-hand *description* of what should be done by the glue, and let the XS compiler **xsubpp** handle the rest.

The XS language allows one to describe the mapping between how the C routine is used, and how the corresponding Perl routine is used. It also allows creation of Perl routines which are directly translated to C code and which are not related to a pre-existing C function. In cases when the C interface coincides with the Perl interface, the XSUB declaration is almost identical to a declaration of a C function (in K&R style). In such circumstances, there is another tool called `h2xs` that is able to translate an entire C header file into a corresponding XS file that will provide glue to the functions/macros described in the header file.

The XS compiler is called **xsubpp**. This compiler creates the constructs necessary to let an XSUB manipulate Perl values, and creates the glue necessary to let Perl call the XSUB. The compiler uses **typemaps** to determine how to map C function parameters and output values to Perl values and back. The default typemap (which comes with Perl) handles many common C types. A supplementary typemap may also be needed to handle any special structures and types for the library being linked.

A file in XS format starts with a C language section which goes until the first `MODULE =` directive. Other XS directives and XSUB definitions may follow this line. The "language" used in this part of the file is usually referred to as the XS language. **xsubpp** recognizes and skips POD (see *perlpod*) in both the C and XS language sections, which allows the XS file to contain embedded documentation.

See *perlxstut* for a tutorial on the whole extension creation process.

Note: For some extensions, Dave Beazley's SWIG system may provide a significantly more convenient mechanism for creating the extension glue code. See http://www.swig.org/ for more information.

On The Road

Many of the examples which follow will concentrate on creating an interface between Perl and the ONC+ RPC bind library functions. The rpcb_gettime() function is used to demonstrate many features of the XS language. This function has two parameters; the first is an input parameter and the second is an output parameter. The function also returns a status value.

```
bool_t rpcb_gettime(const char *host, time_t *timep);
```

From C this function will be called with the following statements.

```
#include <rpc/rpc.h>
bool_t status;
time_t timep;
status = rpcb_gettime( "localhost", &timep );
```

If an XSUB is created to offer a direct translation between this function and Perl, then this XSUB will be used from Perl with the following code. The $status and $timep variables will contain the output of the function.

```
use RPC;
$status = rpcb_gettime( "localhost", $timep );
```

The following XS file shows an XS subroutine, or XSUB, which demonstrates one possible interface to the rpcb_gettime() function. This XSUB represents a direct translation between C and Perl and so preserves the interface even from Perl. This XSUB will be invoked from Perl with the usage shown above. Note that the first three #include statements, for `EXTERN.h`, `perl.h`, and `XSUB.h`, will always be present at the beginning of an XS file. This approach and others will be expanded later in this document.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include <rpc/rpc.h>

MODULE = RPC  PACKAGE = RPC

bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep
   OUTPUT:
     timep
```

Any extension to Perl, including those containing XSUBs, should have a Perl module to serve as the bootstrap which pulls the extension into Perl. This module will export the extension's functions and variables to the Perl program and will cause the extension's XSUBs to be linked into Perl. The following module will be used for most of the examples in this document and should be used from Perl with the `use` command as shown earlier. Perl modules are explained in more detail later in this document.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
```

```
@EXPORT = qw( rpcb_gettime );

bootstrap RPC;
1;
```

Throughout this document a variety of interfaces to the rpcb_gettime() XSUB will be explored. The XSUBs will take their parameters in different orders or will take different numbers of parameters. In each case the XSUB is an abstraction between Perl and the real C rpcb_gettime() function, and the XSUB must always ensure that the real rpcb_gettime() function is called with the correct parameters. This abstraction will allow the programmer to create a more Perl-like interface to the C function.

The Anatomy of an XSUB

The simplest XSUBs consist of 3 parts: a description of the return value, the name of the XSUB routine and the names of its arguments, and a description of types or formats of the arguments.

The following XSUB allows a Perl program to access a C library function called sin(). The XSUB will imitate the C function which takes a single argument and returns a single value.

```
double
sin(x)
  double x
```

Optionally, one can merge the description of types and the list of argument names, rewriting this as

```
double
sin(double x)
```

This makes this XSUB look similar to an ANSI C declaration. An optional semicolon is allowed after the argument list, as in

```
double
sin(double x);
```

Parameters with C pointer types can have different semantic: C functions with similar declarations

```
bool string_looks_as_a_number(char *s);
bool make_char_uppercase(char *c);
```

are used in absolutely incompatible manner. Parameters to these functions could be described **xsubpp** like this:

```
char *  s
char    &c
```

Both these XS declarations correspond to the `char*` C type, but they have different semantics, see *The & Unary Operator*.

It is convenient to think that the indirection operator * should be considered as a part of the type and the address operator & should be considered part of the variable. See *The Typemap* for more info about handling qualifiers and unary operators in C types.

The function name and the return type must be placed on separate lines and should be flush left-adjusted.

```
  INCORRECT                         CORRECT
```

```
double sin(x)                   double
  double x                      sin(x)
       double x
```

The rest of the function description may be indented or left-adjusted. The following example shows a function with its body left-adjusted. Most examples in this document will indent the body for better readability.

```
CORRECT


double
sin(x)
double x
```

More complicated XSUBs may contain many other sections. Each section of an XSUB starts with the corresponding keyword, such as INIT: or CLEANUP:. However, the first two lines of an XSUB always contain the same data: descriptions of the return type and the names of the function and its parameters. Whatever immediately follows these is considered to be an INPUT: section unless explicitly marked with another keyword. (See *The INPUT: Keyword*.)

An XSUB section continues until another section-start keyword is found.

The Argument Stack

The Perl argument stack is used to store the values which are sent as parameters to the XSUB and to store the XSUB's return value(s). In reality all Perl functions (including non-XSUB ones) keep their values on this stack all the same time, each limited to its own range of positions on the stack. In this document the first position on that stack which belongs to the active function will be referred to as position 0 for that function.

XSUBs refer to their stack arguments with the macro **ST(x)**, where *x* refers to a position in this XSUB's part of the stack. Position 0 for that function would be known to the XSUB as ST(0). The XSUB's incoming parameters and outgoing return values always begin at ST(0). For many simple cases the **xsubpp** compiler will generate the code necessary to handle the argument stack by embedding code fragments found in the typemaps. In more complex cases the programmer must supply the code.

The RETVAL Variable

The RETVAL variable is a special C variable that is declared automatically for you. The C type of RETVAL matches the return type of the C library function. The **xsubpp** compiler will declare this variable in each XSUB with non-`void` return type. By default the generated C function will use RETVAL to hold the return value of the C library function being called. In simple cases the value of RETVAL will be placed in ST(0) of the argument stack where it can be received by Perl as the return value of the XSUB.

If the XSUB has a return type of `void` then the compiler will not declare a RETVAL variable for that function. When using a PPCODE: section no manipulation of the RETVAL variable is required, the section may use direct stack manipulation to place output values on the stack.

If PPCODE: directive is not used, `void` return value should be used only for subroutines which do not return a value, *even if* CODE: directive is used which sets ST(0) explicitly.

Older versions of this document recommended to use `void` return value in such cases. It was discovered that this could lead to segfaults in cases when XSUB was *truly* `void`. This practice is now deprecated, and may be not supported at some future version. Use the return value `SV *` in such cases. (Currently `xsubpp` contains some heuristic code which tries to disambiguate between "truly-void" and "old-practice-declared-as-void" functions. Hence your code is at mercy of this heuristics unless you use `SV *` as return value.)

Returning SVs, AVs and HVs through RETVAL

When you're using RETVAL to return an `SV *`, there's some magic going on behind the scenes that should be mentioned. When you're manipulating the argument stack using the ST(x) macro, for example, you usually have to pay special attention to reference counts. (For more about reference counts, see *perlguts*.) To make your life easier, the typemap file automatically makes RETVAL mortal when you're returning an `SV *`. Thus, the following two XSUBs are more or less equivalent:

```
void
alpha()
    PPCODE:
        ST(0) = newSVpv("Hello World",0);
        sv_2mortal(ST(0));
        XSRETURN(1);

SV *
beta()
    CODE:
        RETVAL = newSVpv("Hello World",0);
    OUTPUT:
        RETVAL
```

This is quite useful as it usually improves readability. While this works fine for an `SV *`, it's unfortunately not as easy to have `AV *` or `HV *` as a return value. You *should* be able to write:

```
AV *
array()
    CODE:
        RETVAL = newAV();
        /* do something with RETVAL */
    OUTPUT:
        RETVAL
```

But due to an unfixable bug (fixing it would break lots of existing CPAN modules) in the typemap file, the reference count of the `AV *` is not properly decremented. Thus, the above XSUB would leak memory whenever it is being called. The same problem exists for `HV *`.

When you're returning an `AV *` or a `HV *`, you have make sure their reference count is decremented by making the AV or HV mortal:

```
AV *
array()
    CODE:
        RETVAL = newAV();
        sv_2mortal((SV*)RETVAL);
        /* do something with RETVAL */
    OUTPUT:
        RETVAL
```

And also remember that you don't have to do this for an `SV *`.

The MODULE Keyword

The MODULE keyword is used to start the XS code and to specify the package of the functions which are being defined. All text preceding the first MODULE keyword is considered C code and is passed through to the output with POD stripped, but otherwise untouched. Every XS module will have a bootstrap function which is used to hook the XSUBs into Perl. The package name of this bootstrap function will match the value of the last MODULE statement in the XS source files. The value of

MODULE should always remain constant within the same XS file, though this is not required.

The following example will start the XS code and will place all functions in a package named RPC.

```
MODULE = RPC
```

The PACKAGE Keyword

When functions within an XS source file must be separated into packages the PACKAGE keyword should be used. This keyword is used with the MODULE keyword and must follow immediately after it when used.

```
MODULE = RPC  PACKAGE = RPC

[ XS code in package RPC ]

MODULE = RPC  PACKAGE = RPCB

[ XS code in package RPCB ]

MODULE = RPC  PACKAGE = RPC

[ XS code in package RPC ]
```

The same package name can be used more than once, allowing for non-contiguous code. This is useful if you have a stronger ordering principle than package names.

Although this keyword is optional and in some cases provides redundant information it should always be used. This keyword will ensure that the XSUBs appear in the desired package.

The PREFIX Keyword

The PREFIX keyword designates prefixes which should be removed from the Perl function names. If the C function is `rpcb_gettime()` and the PREFIX value is `rpcb_` then Perl will see this function as `gettime()`.

This keyword should follow the PACKAGE keyword when used. If PACKAGE is not used then PREFIX should follow the MODULE keyword.

```
MODULE = RPC  PREFIX = rpc_

MODULE = RPC  PACKAGE = RPCB  PREFIX = rpcb_
```

The OUTPUT: Keyword

The OUTPUT: keyword indicates that certain function parameters should be updated (new values made visible to Perl) when the XSUB terminates or that certain values should be returned to the calling Perl function. For simple functions which have no CODE: or PPCODE: section, such as the sin() function above, the RETVAL variable is automatically designated as an output value. For more complex functions the **xsubpp** compiler will need help to determine which variables are output variables.

This keyword will normally be used to complement the CODE: keyword. The RETVAL variable is not recognized as an output variable when the CODE: keyword is present. The OUTPUT: keyword is used in this situation to tell the compiler that RETVAL really is an output variable.

The OUTPUT: keyword can also be used to indicate that function parameters are output variables. This may be necessary when a parameter has been modified within the function and the programmer

would like the update to be seen by Perl.

```
bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep
   OUTPUT:
     timep
```

The OUTPUT: keyword will also allow an output parameter to be mapped to a matching piece of code rather than to a typemap.

```
bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep
   OUTPUT:
     timep sv_setnv(ST(1), (double)timep);
```

xsubpp emits an automatic `SvSETMAGIC()` for all parameters in the OUTPUT section of the XSUB, except RETVAL. This is the usually desired behavior, as it takes care of properly invoking 'set' magic on output parameters (needed for hash or array element parameters that must be created if they didn't exist). If for some reason, this behavior is not desired, the OUTPUT section may contain a `SETMAGIC: DISABLE` line to disable it for the remainder of the parameters in the OUTPUT section. Likewise, `SETMAGIC: ENABLE` can be used to reenable it for the remainder of the OUTPUT section. See *perlguts* for more details about 'set' magic.

The NO_OUTPUT Keyword

The NO_OUTPUT can be placed as the first token of the XSUB. This keyword indicates that while the C subroutine we provide an interface to has a non-`void` return type, the return value of this C subroutine should not be returned from the generated Perl subroutine.

With this keyword present *The RETVAL Variable* is created, and in the generated call to the subroutine this variable is assigned to, but the value of this variable is not going to be used in the auto-generated code.

This keyword makes sense only if `RETVAL` is going to be accessed by the user-supplied code. It is especially useful to make a function interface more Perl-like, especially when the C return value is just an error condition indicator. For example,

```
NO_OUTPUT int
delete_file(char *name)
  POSTCALL:
    if (RETVAL != 0)
 croak("Error %d while deleting file '%s'", RETVAL, name);
```

Here the generated XS function returns nothing on success, and will die() with a meaningful error message on error.

The CODE: Keyword

This keyword is used in more complicated XSUBs which require special handling for the C function. The RETVAL variable is still declared, but it will not be returned unless it is specified in the OUTPUT: section.

The following XSUB is for a C function which requires special handling of its parameters. The Perl usage is given first.

```
    $status = rpcb_gettime( "localhost", $timep );
```

The XSUB follows.

```
    bool_t
    rpcb_gettime(host,timep)
         char *host
         time_t timep
       CODE:
              RETVAL = rpcb_gettime( host, &timep );
       OUTPUT:
         timep
         RETVAL
```

The INIT: Keyword

The INIT: keyword allows initialization to be inserted into the XSUB before the compiler generates the call to the C function. Unlike the CODE: keyword above, this keyword does not affect the way the compiler handles RETVAL.

```
   bool_t
   rpcb_gettime(host,timep)
         char *host
         time_t &timep
INIT:
  printf("# Host is %s\n", host );
         OUTPUT:
         timep
```

Another use for the INIT: section is to check for preconditions before making a call to the C function:

```
   long long
   lldiv(a,b)
long long a
long long b
     INIT:
if (a == 0 && b == 0)
    XSRETURN_UNDEF;
if (b == 0)
    croak("lldiv: cannot divide by 0");
```

The NO_INIT Keyword

The NO_INIT keyword is used to indicate that a function parameter is being used only as an output value. The **xsubpp** compiler will normally generate code to read the values of all function parameters from the argument stack and assign them to C variables upon entry to the function. NO_INIT will tell the compiler that some parameters will be used for output rather than for input and that they will be handled before the function terminates.

The following example shows a variation of the rpcb_gettime() function. This function uses the timep variable only as an output variable and does not care about its initial contents.

```
    bool_t
    rpcb_gettime(host,timep)
         char *host
         time_t &timep = NO_INIT
       OUTPUT:
         timep
```

Initializing Function Parameters

C function parameters are normally initialized with their values from the argument stack (which in turn contains the parameters that were passed to the XSUB from Perl). The typemaps contain the code segments which are used to translate the Perl values to the C parameters. The programmer, however, is allowed to override the typemaps and supply alternate (or additional) initialization code. Initialization code starts with the first =, `;` or + on a line in the INPUT: section. The only exception happens if this `;` terminates the line, then this `;` is quietly ignored.

The following code demonstrates how to supply initialization code for function parameters. The initialization code is eval'ed within double quotes by the compiler before it is added to the output so anything which should be interpreted literally [mainly $, @, or \\] must be protected with backslashes. The variables $var, $arg, and $type can be used as in typemaps.

```
bool_t
rpcb_gettime(host,timep)
     char *host = (char *)SvPV($arg,PL_na);
     time_t &timep = 0;
   OUTPUT:
     timep
```

This should not be used to supply default values for parameters. One would normally use this when a function parameter must be processed by another library function before it can be used. Default parameters are covered in the next section.

If the initialization begins with =, then it is output in the declaration for the input variable, replacing the initialization supplied by the typemap. If the initialization begins with `;` or +, then it is performed after all of the input variables have been declared. In the `;` case the initialization normally supplied by the typemap is not performed. For the + case, the declaration for the variable will include the initialization from the typemap. A global variable, %v, is available for the truly rare case where information from one initialization is needed in another initialization.

Here's a truly obscure example:

```
bool_t
rpcb_gettime(host,timep)
     time_t &timep; /* \$v{timep}=@{[$v{timep}=$arg]} */
     char *host + SvOK($v{timep}) ? SvPV($arg,PL_na) : NULL;
   OUTPUT:
     timep
```

The construct `\$v{timep}=@{[$v{timep}=$arg]}` used in the above example has a two-fold purpose: first, when this line is processed by **xsubpp**, the Perl snippet `$v{timep}=$arg` is evaluated. Second, the text of the evaluated snippet is output into the generated C file (inside a C comment)! During the processing of `char *host` line, $arg will evaluate to `ST(0)`, and `$v{timep}` will evaluate to `ST(1)`.

Default Parameter Values

Default values for XSUB arguments can be specified by placing an assignment statement in the parameter list. The default value may be a number, a string or the special string `NO_INIT`. Defaults should always be used on the right-most parameters only.

To allow the XSUB for rpcb_gettime() to have a default host value the parameters to the XSUB could be rearranged. The XSUB will then call the real rpcb_gettime() function with the parameters in the correct order. This XSUB can be called from Perl with either of the following statements:

```
$status = rpcb_gettime( $timep, $host );
```

```
    $status = rpcb_gettime( $timep );
```

The XSUB will look like the code which follows. A CODE: block is used to call the real rpcb_gettime() function with the parameters in the correct order for that function.

```
    bool_t
    rpcb_gettime(timep,host="localhost")
         char *host
         time_t timep = NO_INIT
       CODE:
              RETVAL = rpcb_gettime( host, &timep );
       OUTPUT:
         timep
         RETVAL
```

The PREINIT: Keyword

The PREINIT: keyword allows extra variables to be declared immediately before or after the declarations of the parameters from the INPUT: section are emitted.

If a variable is declared inside a CODE: section it will follow any typemap code that is emitted for the input parameters. This may result in the declaration ending up after C code, which is C syntax error. Similar errors may happen with an explicit `;`-type or +-type initialization of parameters is used (see *Initializing Function Parameters*). Declaring these variables in an INIT: section will not help.

In such cases, to force an additional variable to be declared together with declarations of other variables, place the declaration into a PREINIT: section. The PREINIT: keyword may be used one or more times within an XSUB.

The following examples are equivalent, but if the code is using complex typemaps then the first example is safer.

```
    bool_t
    rpcb_gettime(timep)
         time_t timep = NO_INIT
PREINIT:
         char *host = "localhost";
       CODE:
  RETVAL = rpcb_gettime( host, &timep );
       OUTPUT:
         timep
         RETVAL
```

For this particular case an INIT: keyword would generate the same C code as the PREINIT: keyword. Another correct, but error-prone example:

```
    bool_t
    rpcb_gettime(timep)
         time_t timep = NO_INIT
CODE:
         char *host = "localhost";
  RETVAL = rpcb_gettime( host, &timep );
       OUTPUT:
         timep
         RETVAL
```

Another way to declare `host` is to use a C block in the CODE: section:

```
    bool_t
    rpcb_gettime(timep)
         time_t timep = NO_INIT
CODE:
  {
           char *host = "localhost";
    RETVAL = rpcb_gettime( host, &timep );
  }
      OUTPUT:
        timep
        RETVAL
```

The ability to put additional declarations before the typemap entries are processed is very handy in the cases when typemap conversions manipulate some global state:

```
   MyObject
   mutate(o)
PREINIT:
    MyState st = global_state;
INPUT:
    MyObject o;
CLEANUP:
    reset_to(global_state, st);
```

Here we suppose that conversion to `MyObject` in the INPUT: section and from MyObject when processing RETVAL will modify a global variable `global_state`. After these conversions are performed, we restore the old value of `global_state` (to avoid memory leaks, for example).

There is another way to trade clarity for compactness: INPUT sections allow declaration of C variables which do not appear in the parameter list of a subroutine. Thus the above code for mutate() can be rewritten as

```
   MyObject
   mutate(o)
  MyState st = global_state;
  MyObject o;
CLEANUP:
  reset_to(global_state, st);
```

and the code for rpcb_gettime() can be rewritten as

```
    bool_t
    rpcb_gettime(timep)
  time_t timep = NO_INIT
  char *host = "localhost";
C_ARGS:
  host, &timep
OUTPUT:
         timep
         RETVAL
```

The SCOPE: Keyword

The SCOPE: keyword allows scoping to be enabled for a particular XSUB. If enabled, the XSUB will invoke ENTER and LEAVE automatically.

To support potentially complex type mappings, if a typemap entry used by an XSUB contains a

comment like `/*scope*/` then scoping will be automatically enabled for that XSUB.

To enable scoping:

```
    SCOPE: ENABLE
```

To disable scoping:

```
    SCOPE: DISABLE
```

The INPUT: Keyword

The XSUB's parameters are usually evaluated immediately after entering the XSUB. The INPUT: keyword can be used to force those parameters to be evaluated a little later. The INPUT: keyword can be used multiple times within an XSUB and can be used to list one or more input variables. This keyword is used with the PREINIT: keyword.

The following example shows how the input parameter `timep` can be evaluated late, after a PREINIT.

```
    bool_t
    rpcb_gettime(host,timep)
          char *host
 PREINIT:
   time_t tt;
 INPUT:
          time_t timep
        CODE:
               RETVAL = rpcb_gettime( host, &tt );
        timep = tt;
        OUTPUT:
          timep
          RETVAL
```

The next example shows each input parameter evaluated late.

```
    bool_t
    rpcb_gettime(host,timep)
 PREINIT:
   time_t tt;
 INPUT:
          char *host
 PREINIT:
   char *h;
 INPUT:
          time_t timep
        CODE:
        h = host;
        RETVAL = rpcb_gettime( h, &tt );
        timep = tt;
        OUTPUT:
          timep
          RETVAL
```

Since INPUT sections allow declaration of C variables which do not appear in the parameter list of a subroutine, this may be shortened to:

```
    bool_t
    rpcb_gettime(host,timep)
```

```
time_t tt;
       char *host;
char *h = host;
       time_t timep;
     CODE:
RETVAL = rpcb_gettime( h, &tt );
timep = tt;
     OUTPUT:
       timep
       RETVAL
```

(We used our knowledge that input conversion for `char *` is a "simple" one, thus `host` is initialized on the declaration line, and our assignment `h = host` is not performed too early. Otherwise one would need to have the assignment `h = host` in a CODE: or INIT: section.)

The IN/OUTLIST/IN_OUTLIST/OUT/IN_OUT Keywords

In the list of parameters for an XSUB, one can precede parameter names by the `IN`/`OUTLIST`/`IN_OUTLIST`/`OUT`/`IN_OUT` keywords. `IN` keyword is the default, the other keywords indicate how the Perl interface should differ from the C interface.

Parameters preceded by `OUTLIST`/`IN_OUTLIST`/`OUT`/`IN_OUT` keywords are considered to be used by the C subroutine *via pointers*. `OUTLIST`/`OUT` keywords indicate that the C subroutine does not inspect the memory pointed by this parameter, but will write through this pointer to provide additional return values.

Parameters preceded by `OUTLIST` keyword do not appear in the usage signature of the generated Perl function.

Parameters preceded by `IN_OUTLIST`/`IN_OUT`/`OUT` *do* appear as parameters to the Perl function. With the exception of `OUT`-parameters, these parameters are converted to the corresponding C type, then pointers to these data are given as arguments to the C function. It is expected that the C function will write through these pointers.

The return list of the generated Perl function consists of the C return value from the function (unless the XSUB is of `void` return type or `The NO_OUTPUT Keyword` was used) followed by all the `OUTLIST` and `IN_OUTLIST` parameters (in the order of appearance). On the return from the XSUB the `IN_OUT`/`OUT` Perl parameter will be modified to have the values written by the C function.

For example, an XSUB

```
void
day_month(OUTLIST day, IN unix_time, OUTLIST month)
  int day
  int unix_time
  int month
```

should be used from Perl as

```
my ($day, $month) = day_month(time);
```

The C signature of the corresponding function should be

```
void day_month(int *day, int unix_time, int *month);
```

The `IN`/`OUTLIST`/`IN_OUTLIST`/`IN_OUT`/`OUT` keywords can be mixed with ANSI-style declarations, as in

```
void
```

```
day_month(OUTLIST int day, int unix_time, OUTLIST int month)
```

(here the optional `IN` keyword is omitted).

The `IN_OUT` parameters are identical with parameters introduced with *The & Unary Operator* and put into the `OUTPUT:` section (see *The OUTPUT: Keyword*). The `IN_OUTLIST` parameters are very similar, the only difference being that the value C function writes through the pointer would not modify the Perl parameter, but is put in the output list.

The `OUTLIST`/`OUT` parameter differ from `IN_OUTLIST`/`IN_OUT` parameters only by the initial value of the Perl parameter not being read (and not being given to the C function - which gets some garbage instead). For example, the same C function as above can be interfaced with as

```
void day_month(OUT int day, int unix_time, OUT int month);
```

or

```
void
day_month(day, unix_time, month)
    int &day = NO_INIT
    int  unix_time
    int &month = NO_INIT
  OUTPUT:
    day
    month
```

However, the generated Perl function is called in very C-ish style:

```
my ($day, $month);
day_month($day, time, $month);
```

The length(NAME) Keyword

If one of the input arguments to the C function is the length of a string argument `NAME`, one can substitute the name of the length-argument by `length(NAME)` in the XSUB declaration. This argument must be omitted when the generated Perl function is called. E.g.,

```
void
dump_chars(char *s, short l)
{
  short n = 0;
  while (n < l) {
      printf("s[%d] = \"\\%#03o\"\n", n, (int)s[n]);
      n++;
  }
}

MODULE = x  PACKAGE = x

void dump_chars(char *s, short length(s))
```

should be called as `dump_chars($string)`.

This directive is supported with ANSI-type function declarations only.

Variable-length Parameter Lists

XSUBs can have variable-length parameter lists by specifying an ellipsis (`...`) in the parameter list. This use of the ellipsis is similar to that found in ANSI C. The programmer is able to determine the number of arguments passed to the XSUB by examining the `items` variable which the **xsubpp** compiler supplies for all XSUBs. By using this mechanism one can create an XSUB which accepts a list of parameters of unknown length.

The *host* parameter for the rpcb_gettime() XSUB can be optional so the ellipsis can be used to indicate that the XSUB will take a variable number of parameters. Perl should be able to call this XSUB with either of the following statements.

```
    $status = rpcb_gettime( $timep, $host );

    $status = rpcb_gettime( $timep );
```

The XS code, with ellipsis, follows.

```
    bool_t
    rpcb_gettime(timep, ...)
         time_t timep = NO_INIT
PREINIT:
         char *host = "localhost";
  STRLEN n_a;
     CODE:
  if( items > 1 )
       host = (char *)SvPV(ST(1), n_a);
  RETVAL = rpcb_gettime( host, &timep );
       OUTPUT:
         timep
         RETVAL
```

The C_ARGS: Keyword

The C_ARGS: keyword allows creating of XSUBS which have different calling sequence from Perl than from C, without a need to write CODE: or PPCODE: section. The contents of the C_ARGS: paragraph is put as the argument to the called C function without any change.

For example, suppose that a C function is declared as

```
    symbolic nth_derivative(int n, symbolic function, int flags);
```

and that the default flags are kept in a global C variable `default_flags`. Suppose that you want to create an interface which is called as

```
    $second_deriv = $function->nth_derivative(2);
```

To do this, declare the XSUB as

```
    symbolic
    nth_derivative(function, n)
symbolic function
int  n
     C_ARGS:
n, function, default_flags
```

The PPCODE: Keyword

The PPCODE: keyword is an alternate form of the CODE: keyword and is used to tell the **xsubpp** compiler that the programmer is supplying the code to control the argument stack for the XSUBs return values. Occasionally one will want an XSUB to return a list of values rather than a single value. In these cases one must use PPCODE: and then explicitly push the list of values on the stack. The PPCODE: and CODE: keywords should not be used together within the same XSUB.

The actual difference between PPCODE: and CODE: sections is in the initialization of SP macro (which stands for the *current* Perl stack pointer), and in the handling of data on the stack when returning from an XSUB. In CODE: sections SP preserves the value which was on entry to the XSUB: SP is on the function pointer (which follows the last parameter). In PPCODE: sections SP is moved backward to the beginning of the parameter list, which allows `PUSH*()` macros to place output values in the place Perl expects them to be when the XSUB returns back to Perl.

The generated trailer for a CODE: section ensures that the number of return values Perl will see is either 0 or 1 (depending on the `void`ness of the return value of the C function, and heuristics mentioned in *The RETVAL Variable*). The trailer generated for a PPCODE: section is based on the number of return values and on the number of times `SP` was updated by `[X]PUSH*()` macros.

Note that macros `ST(i)`, `XST_m*()` and `XSRETURN*()` work equally well in CODE: sections and PPCODE: sections.

The following XSUB will call the C rpcb_gettime() function and will return its two output values, timep and status, to Perl as a single list.

```
     void
     rpcb_gettime(host)
          char *host
 PREINIT:
          time_t  timep;
          bool_t  status;
        PPCODE:
          status = rpcb_gettime( host, &timep );
          EXTEND(SP, 2);
          PUSHs(sv_2mortal(newSViv(status)));
          PUSHs(sv_2mortal(newSViv(timep)));
```

Notice that the programmer must supply the C code necessary to have the real rpcb_gettime() function called and to have the return values properly placed on the argument stack.

The `void` return type for this function tells the **xsubpp** compiler that the RETVAL variable is not needed or used and that it should not be created. In most scenarios the void return type should be used with the PPCODE: directive.

The EXTEND() macro is used to make room on the argument stack for 2 return values. The PPCODE: directive causes the **xsubpp** compiler to create a stack pointer available as `SP`, and it is this pointer which is being used in the EXTEND() macro. The values are then pushed onto the stack with the PUSHs() macro.

Now the rpcb_gettime() function can be used from Perl with the following statement.

```
     ($status, $timep) = rpcb_gettime("localhost");
```

When handling output parameters with a PPCODE section, be sure to handle 'set' magic properly. See *perlguts* for details about 'set' magic.

Returning Undef And Empty Lists

Occasionally the programmer will want to return simply `undef` or an empty list if a function fails rather than a separate status value. The rpcb_gettime() function offers just this situation. If the function succeeds we would like to have it return the time and if it fails we would like to have undef returned. In the following Perl code the value of $timep will either be undef or it will be a valid time.

```
$timep = rpcb_gettime( "localhost" );
```

The following XSUB uses the SV * return type as a mnemonic only, and uses a CODE: block to indicate to the compiler that the programmer has supplied all the necessary code. The sv_newmortal() call will initialize the return value to undef, making that the default return value.

```
SV *
rpcb_gettime(host)
     char *  host
PREINIT:
     time_t  timep;
     bool_t x;
   CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) )
          sv_setnv( ST(0), (double)timep);
```

The next example demonstrates how one would place an explicit undef in the return value, should the need arise.

```
SV *
rpcb_gettime(host)
     char *  host
PREINIT:
     time_t  timep;
     bool_t x;
   CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) ){
          sv_setnv( ST(0), (double)timep);
     }
     else{
          ST(0) = &PL_sv_undef;
     }
```

To return an empty list one must use a PPCODE: block and then not push return values on the stack.

```
void
rpcb_gettime(host)
     char *host
PREINIT:
     time_t  timep;
   PPCODE:
     if( rpcb_gettime( host, &timep ) )
          PUSHs(sv_2mortal(newSViv(timep)));
     else{
   /* Nothing pushed on stack, so an empty
    * list is implicitly returned. */
     }
```

Some people may be inclined to include an explicit `return` in the above XSUB, rather than letting control fall through to the end. In those situations `XSRETURN_EMPTY` should be used, instead. This will ensure that the XSUB stack is properly adjusted. Consult *perlapi* for other `XSRETURN` macros.

Since `XSRETURN_*` macros can be used with CODE blocks as well, one can rewrite this example as:

```
     int
     rpcb_gettime(host)
          char *host
 PREINIT:
          time_t  timep;
        CODE:
          RETVAL = rpcb_gettime( host, &timep );
   if (RETVAL == 0)
  XSRETURN_UNDEF;
 OUTPUT:
   RETVAL
```

In fact, one can put this check into a POSTCALL: section as well. Together with PREINIT: simplifications, this leads to:

```
     int
     rpcb_gettime(host)
          char *host
          time_t  timep;
 POSTCALL:
   if (RETVAL == 0)
  XSRETURN_UNDEF;
```

The REQUIRE: Keyword

The REQUIRE: keyword is used to indicate the minimum version of the **xsubpp** compiler needed to compile the XS module. An XS module which contains the following statement will compile with only **xsubpp** version 1.922 or greater:

```
 REQUIRE: 1.922
```

The CLEANUP: Keyword

This keyword can be used when an XSUB requires special cleanup procedures before it terminates. When the CLEANUP: keyword is used it must follow any CODE:, PPCODE:, or OUTPUT: blocks which are present in the XSUB. The code specified for the cleanup block will be added as the last statements in the XSUB.

The POSTCALL: Keyword

This keyword can be used when an XSUB requires special procedures executed after the C subroutine call is performed. When the POSTCALL: keyword is used it must precede OUTPUT: and CLEANUP: blocks which are present in the XSUB.

See examples in *The NO_OUTPUT Keyword* and *Returning Undef And Empty Lists*.

The POSTCALL: block does not make a lot of sense when the C subroutine call is supplied by user by providing either CODE: or PPCODE: section.

The BOOT: Keyword

The BOOT: keyword is used to add code to the extension's bootstrap function. The bootstrap function is generated by the **xsubpp** compiler and normally holds the statements necessary to register any XSUBs with Perl. With the BOOT: keyword the programmer can tell the compiler to add extra

statements to the bootstrap function.

This keyword may be used any time after the first MODULE keyword and should appear on a line by itself. The first blank line after the keyword will terminate the code block.

```
BOOT:
# The following message will be printed when the
# bootstrap function executes.
printf("Hello from the bootstrap!\n");
```

The VERSIONCHECK: Keyword

The VERSIONCHECK: keyword corresponds to **xsubpp**'s `-versioncheck` and `-noversioncheck` options. This keyword overrides the command line options. Version checking is enabled by default. When version checking is enabled the XS module will attempt to verify that its version matches the version of the PM module.

To enable version checking:

```
VERSIONCHECK: ENABLE
```

To disable version checking:

```
VERSIONCHECK: DISABLE
```

The PROTOTYPES: Keyword

The PROTOTYPES: keyword corresponds to **xsubpp**'s `-prototypes` and `-noprototypes` options. This keyword overrides the command line options. Prototypes are enabled by default. When prototypes are enabled XSUBs will be given Perl prototypes. This keyword may be used multiple times in an XS module to enable and disable prototypes for different parts of the module.

To enable prototypes:

```
PROTOTYPES: ENABLE
```

To disable prototypes:

```
PROTOTYPES: DISABLE
```

The PROTOTYPE: Keyword

This keyword is similar to the PROTOTYPES: keyword above but can be used to force **xsubpp** to use a specific prototype for the XSUB. This keyword overrides all other prototype options and keywords but affects only the current XSUB. Consult *"Prototypes" in perlsub* for information about Perl prototypes.

```
    bool_t
    rpcb_gettime(timep, ...)
          time_t timep = NO_INIT
 PROTOTYPE: $;$
 PREINIT:
          char *host = "localhost";
   STRLEN n_a;
        CODE:
    if( items > 1 )
         host = (char *)SvPV(ST(1), n_a);
    RETVAL = rpcb_gettime( host, &timep );
        OUTPUT:
```

```
        timep
        RETVAL
```

If the prototypes are enabled, you can disable it locally for a given XSUB as in the following example:

```
    void
    rpcb_gettime_noproto()
        PROTOTYPE: DISABLE
    ...
```

The ALIAS: Keyword

The ALIAS: keyword allows an XSUB to have two or more unique Perl names and to know which of those names was used when it was invoked. The Perl names may be fully-qualified with package names. Each alias is given an index. The compiler will setup a variable called `ix` which contain the index of the alias which was used. When the XSUB is called with its declared name `ix` will be 0.

The following example will create aliases `FOO::gettime()` and `BAR::getit()` for this function.

```
    bool_t
    rpcb_gettime(host,timep)
          char *host
          time_t &timep
ALIAS:
    FOO::gettime = 1
    BAR::getit = 2
INIT:
  printf("# ix = %d\n", ix );
       OUTPUT:
          timep
```

The OVERLOAD: Keyword

Instead of writing an overloaded interface using pure Perl, you can also use the OVERLOAD keyword to define additional Perl names for your functions (like the ALIAS: keyword above). However, the overloaded functions must be defined with three parameters (except for the nomethod() function which needs four parameters). If any function has the OVERLOAD: keyword, several additional lines will be defined in the c file generated by xsubpp in order to register with the overload magic.

Since blessed objects are actually stored as RV's, it is useful to use the typemap features to preprocess parameters and extract the actual SV stored within the blessed RV. See the sample for T_PTROBJ_SPECIAL below.

To use the OVERLOAD: keyword, create an XS function which takes three input parameters (or use the c style '...' definition) like this:

```
    SV *
    cmp (lobj, robj, swap)
    My_Module_obj    lobj
    My_Module_obj    robj
    IV               swap
    OVERLOAD: cmp <=>
    { /* function defined here */}
```

In this case, the function will overload both of the three way comparison operators. For all overload operations using non-alpha characters, you must type the parameter without quoting, separating multiple overloads with whitespace. Note that "" (the stringify overload) should be entered as \"\" (i.e. escaped).

The FALLBACK: Keyword

In addition to the OVERLOAD keyword, if you need to control how Perl autogenerates missing overloaded operators, you can set the FALLBACK keyword in the module header section, like this:

```
MODULE = RPC  PACKAGE = RPC

FALLBACK: TRUE
...
```

where FALLBACK can take any of the three values TRUE, FALSE, or UNDEF. If you do not set any FALLBACK value when using OVERLOAD, it defaults to UNDEF. FALLBACK is not used except when one or more functions using OVERLOAD have been defined. Please see *"Fallback" in overload* for more details.

The INTERFACE: Keyword

This keyword declares the current XSUB as a keeper of the given calling signature. If some text follows this keyword, it is considered as a list of functions which have this signature, and should be attached to the current XSUB.

For example, if you have 4 C functions multiply(), divide(), add(), subtract() all having the signature:

```
symbolic f(symbolic, symbolic);
```

you can make them all to use the same XSUB using this:

```
    symbolic
    interface_s_ss(arg1, arg2)
symbolic arg1
symbolic arg2
    INTERFACE:
multiply divide
add subtract
```

(This is the complete XSUB code for 4 Perl functions!) Four generated Perl function share names with corresponding C functions.

The advantage of this approach comparing to ALIAS: keyword is that there is no need to code a switch statement, each Perl function (which shares the same XSUB) knows which C function it should call. Additionally, one can attach an extra function remainder() at runtime by using

```
CV *mycv = newXSproto("Symbolic::remainder",
 XS_Symbolic_interface_s_ss, __FILE__, "$$");
XSINTERFACE_FUNC_SET(mycv, remainder);
```

say, from another XSUB. (This example supposes that there was no INTERFACE_MACRO: section, otherwise one needs to use something else instead of `XSINTERFACE_FUNC_SET`, see the next section.)

The INTERFACE_MACRO: Keyword

This keyword allows one to define an INTERFACE using a different way to extract a function pointer from an XSUB. The text which follows this keyword should give the name of macros which would extract/set a function pointer. The extractor macro is given return type, `CV*`, and `XSANY.any_dptr` for this `CV*`. The setter macro is given cv, and the function pointer.

The default value is `XSINTERFACE_FUNC` and `XSINTERFACE_FUNC_SET`. An INTERFACE keyword with an empty list of functions can be omitted if INTERFACE_MACRO keyword is used.

Suppose that in the previous example functions pointers for multiply(), divide(), add(), subtract() are kept in a global C array `fp[]` with offsets being `multiply_off`, `divide_off`, `add_off`, `subtract_off`. Then one can use

```
    #define XSINTERFACE_FUNC_BYOFFSET(ret,cv,f) \
 ((XSINTERFACE_CVT_ANON(ret))fp[CvXSUBANY(cv).any_i32])
    #define XSINTERFACE_FUNC_BYOFFSET_set(cv,f) \
 CvXSUBANY(cv).any_i32 = CAT2( f, _off )
```

in C section,

```
    symbolic
    interface_s_ss(arg1, arg2)
 symbolic arg1
 symbolic arg2
      INTERFACE_MACRO:
 XSINTERFACE_FUNC_BYOFFSET
 XSINTERFACE_FUNC_BYOFFSET_set
      INTERFACE:
 multiply divide
 add subtract
```

in XSUB section.

The INCLUDE: Keyword

This keyword can be used to pull other files into the XS module. The other files may have XS code. INCLUDE: can also be used to run a command to generate the XS code to be pulled into the module.

The file *Rpcb1.xsh* contains our `rpcb_gettime()` function:

```
    bool_t
    rpcb_gettime(host,timep)
          char *host
          time_t &timep
        OUTPUT:
          timep
```

The XS module can use INCLUDE: to pull that file into it.

```
    INCLUDE: Rpcb1.xsh
```

If the parameters to the INCLUDE: keyword are followed by a pipe (`|`) then the compiler will interpret the parameters as a command.

```
    INCLUDE: cat Rpcb1.xsh |
```

The CASE: Keyword

The CASE: keyword allows an XSUB to have multiple distinct parts with each part acting as a virtual XSUB. CASE: is greedy and if it is used then all other XS keywords must be contained within a CASE:. This means nothing may precede the first CASE: in the XSUB and anything following the last CASE: is included in that case.

A CASE: might switch via a parameter of the XSUB, via the `ix` ALIAS: variable (see *The ALIAS: Keyword*), or maybe via the `items` variable (see *Variable-length Parameter Lists*). The last CASE: becomes the **default** case if it is not associated with a conditional. The following example shows CASE switched via `ix` with a function `rpcb_gettime()` having an alias `x_gettime()`. When the

function is called as `rpcb_gettime()` its parameters are the usual `(char *host, time_t *timep)`, but when the function is called as `x_gettime()` its parameters are reversed, `(time_t *timep, char *host)`.

```
    long
    rpcb_gettime(a,b)
      CASE: ix == 1
 ALIAS:
   x_gettime = 1
 INPUT:
   # 'a' is timep, 'b' is host
         char *b
         time_t a = NO_INIT
       CODE:
             RETVAL = rpcb_gettime( b, &a );
       OUTPUT:
         a
         RETVAL
      CASE:
   # 'a' is host, 'b' is timep
         char *a
         time_t &b = NO_INIT
       OUTPUT:
         b
         RETVAL
```

That function can be called with either of the following statements. Note the different argument lists.

```
 $status = rpcb_gettime( $host, $timep );

 $status = x_gettime( $timep, $host );
```

The & Unary Operator

The & unary operator in the INPUT: section is used to tell **xsubpp** that it should convert a Perl value to/from C using the C type to the left of &, but provide a pointer to this value when the C function is called.

This is useful to avoid a CODE: block for a C function which takes a parameter by reference. Typically, the parameter should be not a pointer type (an `int` or `long` but not an `int*` or `long*`).

The following XSUB will generate incorrect C code. The **xsubpp** compiler will turn this into code which calls `rpcb_gettime()` with parameters `(char *host, time_t timep)`, but the real `rpcb_gettime()` wants the `timep` parameter to be of type `time_t*` rather than `time_t`.

```
    bool_t
    rpcb_gettime(host,timep)
          char *host
          time_t timep
        OUTPUT:
          timep
```

That problem is corrected by using the & operator. The **xsubpp** compiler will now turn this into code which calls `rpcb_gettime()` correctly with parameters `(char *host, time_t *timep)`. It does this by carrying the & through, so the function call looks like `rpcb_gettime(host, &timep)`.

```
    bool_t
```

```
rpcb_gettime(host,timep)
      char *host
      time_t &timep
    OUTPUT:
      timep
```

Inserting POD, Comments and C Preprocessor Directives

C preprocessor directives are allowed within BOOT:, PREINIT: INIT:, CODE:, PPCODE:, POSTCALL:, and CLEANUP: blocks, as well as outside the functions. Comments are allowed anywhere after the MODULE keyword. The compiler will pass the preprocessor directives through untouched and will remove the commented lines. POD documentation is allowed at any point, both in the C and XS language sections. POD must be terminated with a `=cut` command; `xsubpp` will exit with an error if it does not. It is very unlikely that human generated C code will be mistaken for POD, as most indenting styles result in whitespace in front of any line starting with =. Machine generated XS files may fall into this trap unless care is taken to ensure that a space breaks the sequence "\n=".

Comments can be added to XSUBs by placing a # as the first non-whitespace of a line. Care should be taken to avoid making the comment look like a C preprocessor directive, lest it be interpreted as such. The simplest way to prevent this is to put whitespace in front of the #.

If you use preprocessor directives to choose one of two versions of a function, use

```
#if ... version1
#else /* ... version2  */
#endif
```

and not

```
#if ... version1
#endif
#if ... version2
#endif
```

because otherwise **xsubpp** will believe that you made a duplicate definition of the function. Also, put a blank line before the #else/#endif so it will not be seen as part of the function body.

Using XS With C++

If an XSUB name contains `::`, it is considered to be a C++ method. The generated Perl function will assume that its first argument is an object pointer. The object pointer will be stored in a variable called THIS. The object should have been created by C++ with the new() function and should be blessed by Perl with the sv_setref_pv() macro. The blessing of the object by Perl can be handled by a typemap. An example typemap is shown at the end of this section.

If the return type of the XSUB includes `static`, the method is considered to be a static method. It will call the C++ function using the class::method() syntax. If the method is not static the function will be called using the THIS->method() syntax.

The next examples will use the following C++ class.

```
 class color {
      public:
      color();
      ~color();
      int blue();
      void set_blue( int );

      private:
```

```
        int c_blue;
};
```

The XSUBs for the blue() and set_blue() methods are defined with the class name but the parameter for the object (THIS, or "self") is implicit and is not listed.

```
int
color::blue()

void
color::set_blue( val )
     int val
```

Both Perl functions will expect an object as the first parameter. In the generated C++ code the object is called `THIS`, and the method call will be performed on this object. So in the C++ code the blue() and set_blue() methods will be called as this:

```
RETVAL = THIS->blue();

THIS->set_blue( val );
```

You could also write a single get/set method using an optional argument:

```
int
color::blue( val = NO_INIT )
    int val
    PROTOTYPE $;$
    CODE:
        if (items > 1)
            THIS->set_blue( val );
        RETVAL = THIS->blue();
    OUTPUT:
        RETVAL
```

If the function's name is **DESTROY** then the C++ `delete` function will be called and `THIS` will be given as its parameter. The generated C++ code for

```
void
color::DESTROY()
```

will look like this:

```
color *THIS = ...; // Initialized as in typemap

delete THIS;
```

If the function's name is **new** then the C++ `new` function will be called to create a dynamic C++ object. The XSUB will expect the class name, which will be kept in a variable called `CLASS`, to be given as the first argument.

```
color *
color::new()
```

The generated C++ code will call `new`.

```
     RETVAL = new color();
```

The following is an example of a typemap that could be used for this C++ example.

```
    TYPEMAP
    color *  O_OBJECT

    OUTPUT
    # The Perl object is blessed into 'CLASS', which should be a
    # char* having the name of the package for the blessing.
    O_OBJECT
     sv_setref_pv( $arg, CLASS, (void*)$var );

    INPUT
    O_OBJECT
     if( sv_isobject($arg) && (SvTYPE(SvRV($arg)) == SVt_PVMG) )
      $var = ($type)SvIV((SV*)SvRV( $arg ));
     else{
      warn( \"${Package}::$func_name() -- $var is not a blessed SV
reference\" );
      XSRETURN_UNDEF;
     }
```

Interface Strategy

When designing an interface between Perl and a C library a straight translation from C to XS (such as created by `h2xs -x`) is often sufficient. However, sometimes the interface will look very C-like and occasionally nonintuitive, especially when the C function modifies one of its parameters, or returns failure inband (as in "negative return values mean failure"). In cases where the programmer wishes to create a more Perl-like interface the following strategy may help to identify the more critical parts of the interface.

Identify the C functions with input/output or output parameters. The XSUBs for these functions may be able to return lists to Perl.

Identify the C functions which use some inband info as an indication of failure. They may be candidates to return undef or an empty list in case of failure. If the failure may be detected without a call to the C function, you may want to use an INIT: section to report the failure. For failures detectable after the C function returns one may want to use a POSTCALL: section to process the failure. In more complicated cases use CODE: or PPCODE: sections.

If many functions use the same failure indication based on the return value, you may want to create a special typedef to handle this situation. Put

```
  typedef int negative_is_failure;
```

near the beginning of XS file, and create an OUTPUT typemap entry for `negative_is_failure` which converts negative values to `undef`, or maybe croak()s. After this the return value of type `negative_is_failure` will create more Perl-like interface.

Identify which values are used by only the C and XSUB functions themselves, say, when a parameter to a function should be a contents of a global variable. If Perl does not need to access the contents of the value then it may not be necessary to provide a translation for that value from C to Perl.

Identify the pointers in the C function parameter lists and return values. Some pointers may be used to implement input/output or output parameters, they can be handled in XS with the `&` unary operator, and, possibly, using the NO_INIT keyword. Some others will require handling of types like `int *`, and

one needs to decide what a useful Perl translation will do in such a case. When the semantic is clear, it is advisable to put the translation into a typemap file.

Identify the structures used by the C functions. In many cases it may be helpful to use the T_PTROBJ typemap for these structures so they can be manipulated by Perl as blessed objects. (This is handled automatically by `h2xs -x`.)

If the same C type is used in several different contexts which require different translations, `typedef` several new types mapped to this C type, and create separate *typemap* entries for these new types. Use these types in declarations of return type and parameters to XSUBs.

Perl Objects And C Structures

When dealing with C structures one should select either **T_PTROBJ** or **T_PTRREF** for the XS type. Both types are designed to handle pointers to complex objects. The T_PTRREF type will allow the Perl object to be unblessed while the T_PTROBJ type requires that the object be blessed. By using T_PTROBJ one can achieve a form of type-checking because the XSUB will attempt to verify that the Perl object is of the expected type.

The following XS code shows the getnetconfigent() function which is used with ONC+ TIRPC. The getnetconfigent() function will return a pointer to a C structure and has the C prototype shown below. The example will demonstrate how the C pointer will become a Perl reference. Perl will consider this reference to be a pointer to a blessed object and will attempt to call a destructor for the object. A destructor will be provided in the XS source to free the memory used by getnetconfigent(). Destructors in XS can be created by specifying an XSUB function whose name ends with the word **DESTROY**. XS destructors can be used to free memory which may have been malloc'd by another XSUB.

```
struct netconfig *getnetconfigent(const char *netid);
```

A `typedef` will be created for `struct netconfig`. The Perl object will be blessed in a class matching the name of the C type, with the tag `Ptr` appended, and the name should not have embedded spaces if it will be a Perl package name. The destructor will be placed in a class corresponding to the class of the object and the PREFIX keyword will be used to trim the name to the word DESTROY as Perl will expect.

```
typedef struct netconfig Netconfig;

MODULE = RPC  PACKAGE = RPC

Netconfig *
getnetconfigent(netid)
     char *netid

MODULE = RPC  PACKAGE = NetconfigPtr  PREFIX = rpcb_

void
rpcb_DESTROY(netconf)
     Netconfig *netconf
   CODE:
     printf("Now in NetconfigPtr::DESTROY\n");
     free( netconf );
```

This example requires the following typemap entry. Consult the typemap section for more information about adding new typemaps for an extension.

```
TYPEMAP
```

```
Netconfig *  T_PTROBJ
```

This example will be used with the following Perl statements.

```
use RPC;
$netconf = getnetconfigent("udp");
```

When Perl destroys the object referenced by $netconf it will send the object to the supplied XSUB DESTROY function. Perl cannot determine, and does not care, that this object is a C struct and not a Perl object. In this sense, there is no difference between the object created by the getnetconfigent() XSUB and an object created by a normal Perl subroutine.

The Typemap

The typemap is a collection of code fragments which are used by the **xsubpp** compiler to map C function parameters and values to Perl values. The typemap file may consist of three sections labelled `TYPEMAP`, `INPUT`, and `OUTPUT`. An unlabelled initial section is assumed to be a `TYPEMAP` section. The INPUT section tells the compiler how to translate Perl values into variables of certain C types. The OUTPUT section tells the compiler how to translate the values from certain C types into values Perl can understand. The TYPEMAP section tells the compiler which of the INPUT and OUTPUT code fragments should be used to map a given C type to a Perl value. The section labels `TYPEMAP`, `INPUT`, or `OUTPUT` must begin in the first column on a line by themselves, and must be in uppercase.

The default typemap in the `lib/ExtUtils` directory of the Perl source contains many useful types which can be used by Perl extensions. Some extensions define additional typemaps which they keep in their own directory. These additional typemaps may reference INPUT and OUTPUT maps in the main typemap. The **xsubpp** compiler will allow the extension's own typemap to override any mappings which are in the default typemap.

Most extensions which require a custom typemap will need only the TYPEMAP section of the typemap file. The custom typemap used in the getnetconfigent() example shown earlier demonstrates what may be the typical use of extension typemaps. That typemap is used to equate a C structure with the T_PTROBJ typemap. The typemap used by getnetconfigent() is shown here. Note that the C type is separated from the XS type with a tab and that the C unary operator `*` is considered to be a part of the C type name.

```
TYPEMAP
Netconfig *<tab>T_PTROBJ
```

Here's a more complicated example: suppose that you wanted `struct netconfig` to be blessed into the class `Net::Config`. One way to do this is to use underscores (_) to separate package names, as follows:

```
typedef struct netconfig * Net_Config;
```

And then provide a typemap entry `T_PTROBJ_SPECIAL` that maps underscores to double-colons (::), and declare `Net_Config` to be of that type:

```
TYPEMAP
Net_Config      T_PTROBJ_SPECIAL

INPUT
T_PTROBJ_SPECIAL
        if (sv_derived_from($arg, \"${(my
$ntt=$ntype)=~s/_/::/g;\$ntt}\")) {
                IV tmp = SvIV((SV*)SvRV($arg));
```

```
                        $var = INT2PTR($type, tmp);
                }
                else
                        croak(\"$var is not of type ${(my
$ntt=$ntype)=~s/_/::/g;\$ntt}\")

        OUTPUT
        T_PTROBJ_SPECIAL
                sv_setref_pv($arg, \"${(my $ntt=$ntype)=~s/_/::/g;\$ntt}\",
                (void*)$var);
```

The INPUT and OUTPUT sections substitute underscores for double-colons on the fly, giving the desired effect. This example demonstrates some of the power and versatility of the typemap facility.

The INT2PTR macro (defined in perl.h) casts an integer to a pointer, of a given type, taking care of the possible different size of integers and pointers. There are also PTR2IV, PTR2UV, PTR2NV macros, to map the other way, which may be useful in OUTPUT sections.

Safely Storing Static Data in XS

Starting with Perl 5.8, a macro framework has been defined to allow static data to be safely stored in XS modules that will be accessed from a multi-threaded Perl.

Although primarily designed for use with multi-threaded Perl, the macros have been designed so that they will work with non-threaded Perl as well.

It is therefore strongly recommended that these macros be used by all XS modules that make use of static data.

The easiest way to get a template set of macros to use is by specifying the `-g` (`--global`) option with h2xs (see *h2xs*).

Below is an example module that makes use of the macros.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

/* Global Data */

#define MY_CXT_KEY "BlindMice::_guts" XS_VERSION

typedef struct {
    int count;
    char name[3][100];
} my_cxt_t;

START_MY_CXT

MODULE = BlindMice           PACKAGE = BlindMice

BOOT:
{
    MY_CXT_INIT;
    MY_CXT.count = 0;
    strcpy(MY_CXT.name[0], "None");
```

```
        strcpy(MY_CXT.name[1], "None");
        strcpy(MY_CXT.name[2], "None");
    }

    int
    newMouse(char * name)
        char * name;
        PREINIT:
          dMY_CXT;
        CODE:
          if (MY_CXT.count >= 3) {
              warn("Already have 3 blind mice");
              RETVAL = 0;
          }
          else {
              RETVAL = ++ MY_CXT.count;
              strcpy(MY_CXT.name[MY_CXT.count - 1], name);
          }

    char *
    get_mouse_name(index)
      int index
      CODE:
        dMY_CXT;
        RETVAL = MY_CXT.lives ++;
        if (index > MY_CXT.count)
          croak("There are only 3 blind mice.");
        else
          RETVAL = newSVpv(MY_CXT.name[index - 1]);

    void
    CLONE(...)
 CODE:
 MY_CXT_CLONE;
```

REFERENCE

MY_CXT_KEY

This macro is used to define a unique key to refer to the static data for an XS module. The suggested naming scheme, as used by h2xs, is to use a string that consists of the module name, the string "::_guts" and the module version number.

```
    #define MY_CXT_KEY "MyModule::_guts" XS_VERSION
```

typedef my_cxt_t

This struct typedef *must* always be called `my_cxt_t` -- the other `CXT*` macros assume the existence of the `my_cxt_t` typedef name.

Declare a typedef named `my_cxt_t` that is a structure that contains all the data that needs to be interpreter-local.

```
    typedef struct {
        int some_value;
    } my_cxt_t;
```

START_MY_CXT

Always place the START_MY_CXT macro directly after the declaration of `my_cxt_t`.

MY_CXT_INIT

The MY_CXT_INIT macro initialises storage for the `my_cxt_t` struct.

It *must* be called exactly once -- typically in a BOOT: section. If you are maintaining multiple interpreters, it should be called once in each interpreter instance, except for interpreters cloned from existing ones. (But see `MY_CXT_CLONE` below.)

dMY_CXT

Use the dMY_CXT macro (a declaration) in all the functions that access MY_CXT.

MY_CXT

Use the MY_CXT macro to access members of the `my_cxt_t` struct. For example, if `my_cxt_t` is

```
    typedef struct {
        int index;
    } my_cxt_t;
```

then use this to access the `index` member

```
    dMY_CXT;
    MY_CXT.index = 2;
```

aMY_CXT/pMY_CXT

`dMY_CXT` may be quite expensive to calculate, and to avoid the overhead of invoking it in each function it is possible to pass the declaration onto other functions using the `aMY_CXT`/`pMY_CXT` macros, eg

```
    void sub1() {
 dMY_CXT;
 MY_CXT.index = 1;
 sub2(aMY_CXT);
    }

    void sub2(pMY_CXT) {
 MY_CXT.index = 2;
    }
```

Analogously to `pTHX`, there are equivalent forms for when the macro is the first or last in multiple arguments, where an underscore represents a comma, i.e. `_aMY_CXT`, `aMY_CXT_`, `_pMY_CXT` and `pMY_CXT_`.

MY_CXT_CLONE

By default, when a new interpreter is created as a copy of an existing one (eg via `<threads-`create()>>), both interpreters share the same physical my_cxt_t structure. Calling `MY_CXT_CLONE` (typically via the package's `CLONE( )` function), causes a byte-for-byte copy of the structure to be taken, and any future dMY_CXT will cause the copy to be accessed instead.

MY_CXT_INIT_INTERP(my_perl)

dMY_CXT_INTERP(my_perl)

These are versions of the macros which take an explicit interpreter as an argument.

Note that these macros will only work together within the *same* source file; that is, a dMY_CTX in one source file will access a different structure than a dMY_CTX in another source file.

Thread-aware system interfaces

Starting from Perl 5.8, in C/C++ level Perl knows how to wrap system/library interfaces that have thread-aware versions (e.g. getpwent_r()) into frontend macros (e.g. getpwent()) that correctly handle the multithreaded interaction with the Perl interpreter. This will happen transparently, the only thing you need to do is to instantiate a Perl interpreter.

This wrapping happens always when compiling Perl core source (PERL_CORE is defined) or the Perl core extensions (PERL_EXT is defined). When compiling XS code outside of Perl core the wrapping does not take place. Note, however, that intermixing the _r-forms (as Perl compiled for multithreaded operation will do) and the _r-less forms is neither well-defined (inconsistent results, data corruption, or even crashes become more likely), nor is it very portable.

EXAMPLES

File `RPC.xs`: Interface to some ONC+ RPC bind library functions.

```
     #include "EXTERN.h"
     #include "perl.h"
     #include "XSUB.h"

     #include <rpc/rpc.h>

     typedef struct netconfig Netconfig;

     MODULE = RPC  PACKAGE = RPC

     SV *
     rpcb_gettime(host="localhost")
          char *host
 PREINIT:
          time_t  timep;
        CODE:
          ST(0) = sv_newmortal();
          if( rpcb_gettime( host, &timep ) )
               sv_setnv( ST(0), (double)timep );

     Netconfig *
     getnetconfigent(netid="udp")
          char *netid

     MODULE = RPC  PACKAGE = NetconfigPtr  PREFIX = rpcb_

     void
     rpcb_DESTROY(netconf)
          Netconfig *netconf
        CODE:
          printf("NetconfigPtr::DESTROY\n");
          free( netconf );
```

File `typemap`: Custom typemap for RPC.xs.

```
     TYPEMAP
     Netconfig *  T_PTROBJ
```

File `RPC.pm`: Perl module for the RPC extension.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw(rpcb_gettime getnetconfigent);

bootstrap RPC;
1;
```

File `rpctest.pl`: Perl test program for the RPC extension.

```
use RPC;

$netconf = getnetconfigent();
$a = rpcb_gettime();
print "time = $a\n";
print "netconf = $netconf\n";

$netconf = getnetconfigent("tcp");
$a = rpcb_gettime("poplar");
print "time = $a\n";
print "netconf = $netconf\n";
```

XS VERSION

This document covers features supported by `xsubpp` 1.935.

AUTHOR

Originally written by Dean Roehrich <*roehrich@cray.com*>.

Maintained since 1996 by The Perl Porters <*perlbug@perl.org*>.