

## Man page headings

Heading	Description
NAME	Name of the program/library/whatever.
SYNOPSIS	Brief example of usage.
DESCRIPTION	Detailed description, broken into sections if necessary.
EXAMPLES	Show us how we use it.
SEE ALSO	References to other man pages, etc.
BUGS	Things that need a little work yet.
AUTHOR	Your name in lights.

See the `pod2man` man page for more information about the layout of man pages.

**Item 47: Use XS for low-level interfaces and/or speed.**

Recent versions of Perl have a documented interface language, called *XS*,<sup>1</sup> that you can use to write functions in C or C++ that can be called from Perl. Within *XSUBs*, as they are called, you have full access to Perl's internals. You can create variables, change their values, execute Perl code, or do pretty much anything that suits your fancy.

An XS module is actually a dynamically loaded shareable library.<sup>2</sup> Creating one is a complex process fraught with details. Fortunately, with the XS interface you get a bunch of tools that handle most of those details for you. For reasonably simple situations, you need only run `x2hs` to get a set of boilerplate files, add some Perl and XS code, and run `make`. There are make targets for building the shareable library, testing it, building a distribution kit, and installing it.

XSUBs are a handy way to add operating-system supported features to Perl—they beat the heck out of `syscall`. You also can use XSUBs to speed up scripts that are spending a significant proportion of their time in a small number of subroutines. And, of course, you can use XSUBs to add a Perl interface to an existing C or C++ library of your own.

---

1. Parts of XS are still under development as of this writing. XS shouldn't change radically in the near future, but don't be surprised to find some differences between your version and what's documented here.

2. If your system does not support shareable libraries, you can still use XSUBs by linking them into the Perl executable itself.

As an example of the sort of things you might write an XSUB for, let's suppose that you would like to write a subroutine that returns a copy of a list, but with all the elements in random order. In Perl, you might write something like:

```
sub shuffle1 {
    my @orig = @_;
    my @result;
    push @result, splice @orig, rand @orig, 1 while @orig;
    @result;
}
```

This isn't exactly a model of efficiency, because you have to call `splice` once for each element in the list—not good if you're planning on shuffling long lists. A more efficient approach might be:

```
sub shuffle2 {
    my @result = @_;
    my $n = @result;
    while ($n > 1) {
        my $i = rand $n;
        $n--;
        @result[$i, $n] = @result[$n, $i];
    }
    @result;
}
```

This is better, but not a lot better. There is no splicing going on, but on the other hand the swapping of values involves a lot of assignments and subscripting, which takes a surprisingly large amount of time. (See the benchmarks at the end of this Item.) In truth, shuffling is best *not* written in Perl if efficiency is a prime concern. So let's write `shuffle` as an XSUB in C.

## Generating the boilerplate

When writing XSUBs, you should always start out by using `h2xs` to generate a set of boilerplate files, much as you would for an ordinary Perl-only module (see Item 45). In this case, let's start by creating boilerplate for a module called `List::Shuffle`.

```
% h2xs -A -n List::Shuffle
Writing List/Shuffle/Shuffle.pm
Writing List/Shuffle/Shuffle.xs
Writing List/Shuffle/Makefile.PL
Writing List/Shuffle/test.pl
Writing List/Shuffle/Changes
Writing List/Shuffle/MANIFEST
```

This is similar to the example in Item 45. However, since we have chosen a hierarchical package name, the boilerplate files are created in a directory named `Shuffle` that is itself nested in a directory named `List`.

As before, you will also need to generate a makefile:

```
% perl Makefile.PL
```

You may want to make some changes to the boilerplate code before you begin writing your XSUB. For example, if you want to be able to call the `shuffle()` subroutine without having to prepend the package name to it, you must export its name in `Shuffle.pm`:

```
@EXPORT = qw(shuffle);
```

## Writing and testing an XSUB

The XS language that XSUBs are written in is really just a kind of C pre-processor designed to make writing XSUBs easier.

XS source code goes into files ending with “.xs”. The XS compiler, `xsubpp`, compiles XS into C code with the glue needed to interface with Perl. You probably will never have to invoke `xsubpp` on your own, though, because it is invoked automatically via the generated makefile.

XS source files begin with a prologue of C code that is passed through unaltered by `xsubpp`. A `MODULE` directive follows the prologue; for example:

```
MODULE = List::Shuffle    PACKAGE = List::Shuffle
```

This indicates the start of the actual XS source, which is itself a list of XSUBs.<sup>3</sup>

So, how about some examples of XS?

Here is a very simple XSUB that just calls the C standard library `log` function and returns the result:

```
double
log(x)
    double x
```

The return type appears first, on a line by itself at the beginning of the line. Next is the function name and a list of parameter names. The lines following the return type and function name are ordinarily indented for

---

3. Unfortunately, a thorough description of how to write XSUBs would fill an entire book (and it probably will, one of these days), and there isn't space to explain XSUBs in great detail here. For now, for detailed information about XS, you will have to rely on the documentation shipped with Perl, notably the `perlxs`, `perlxstut`, and `perlguts` man pages.

readability. In this case, there is only a single line in the XSUB body, declaring the type of the parameter `x`.

In a simple case like this, `xsubpp` generates code that creates a Perl subroutine named `log()` that calls the C function of the same name. The generated code also includes the glue necessary to convert the Perl argument to a C `double` and to convert the result back. (To see how this works, take a look at the C code generated by `xsubpp`—it’s actually pretty readable.)

Here is a slightly more complex example that calls the Unix `realpath()` function (not available on all systems):

```
char *
realpath(filename)
  char *filename
  PREINIT:
    char realname[1024]; /* or use MAXPATHLEN */
  CODE:
    RETVAL = realpath(filename, realname);
  OUTPUT:
    RETVAL
```

This creates a Perl function that takes a string argument and has a string return value. The XS glue takes care of converting Perl strings to C’s `char *` type and back:

```
$realname = realpath($filename);
```

The `CODE` section contains the portion of the code used to compute the result from the subroutine. The `PREINIT` section contains declarations of variables used in the `CODE` section; they should go here rather than in the `CODE` section. `RETVAL` is a “magic” variable supplied by `xsubpp` used to hold the return value. Finally, the `OUTPUT` section lists values that will be returned to the caller. This ordinarily will include `RETVAL`. It can also include input parameters that are modified and returned as if through call by reference.

These examples all return single scalar values. In order to write `shuffle`, which returns a list of scalars, we have to use a `PCODE` section and do our own popping and pushing of arguments and return values. This is less difficult than it may sound. To get things rolling, insert the following code in `Shuffle.xs` following the `MODULE` line:

● XS source code for `List::Shuffle`

```
PROTOTYPES: DISABLE
```

*Turn off Perl prototype processing for the following XSUBs.*

## ● XS source code for List::Shuffle (cont'd)

void	<i>Declared void because we are</i>
shuffle(...)	<i>returning values “manually”</i>
PPCODE:	<i>with XPUSHs.</i>
{	
int i, n;	<i>SV is the “scalar value” type.</i>
SV **array;	
SV *tmp;	
array = New(0, array, items, SV *);	<i>Allocate storage. New()</i>
	<i>requests memory from Perl’s</i>
	<i>memory allocator.</i>
for (i = 0; i < items; i++) {	<i>Copy input args.</i>
array[i] = sv_mortalcopy(ST(i));	
}	
n = items;	
while (n > 1) {	<i>Shuffle off to Buffalo!</i>
i = rand() % n;	
tmp = array[i];	
array[i] = array[--n];	
array[n] = tmp;	
}	
for (i = 0; i < items; i++) {	<i>Push result (a list) onto stack.</i>
XPUSHs(array[i]);	
}	
Safefree(array);	<i>Free storage. Safefree()</i>
}	<i>returns memory to Perl.</i>

The PROTOTYPES: DISABLE directive turns off Perl prototype processing (see Item 28) for the XSUBs that follow.

The strategy here is to copy the input arguments into a temporary array, shuffle them, and then push the result onto the stack. The arguments will be scalar values, which are represented internally in Perl with the type SV \*.

Instead of a CODE block, we use a PPCODE block inside this XSUB. This disables the automatic handling of return values on the stack. The number of arguments passed in is contained in the magic variable items, and the arguments themselves are contained in ST(0), ST(1), and so on.

The SV pointers on the stack refer to the actual values supplied to the shuffle() function. We don’t want this. We want copies instead, so we use the function sv\_mortalcopy() to make a reference counted clone of each incoming scalar.

The scalars go into an array allocated with Perl's internal `New()` function, and then we shuffle them. After shuffling, we push the return values on the stack one at a time with the `XPUSHs()` function and free the temporary storage we used to hold the array of pointers. If all this seems sketchy, which it probably does, consult the `perlguits` and `perlx`s man pages for more details.

At this point, you can save `Shuffle.xs` and build it:

```
% make
```

You will see a few lines of gobbledygook as `Shuffle.xs` compiles (hopefully) and as some other files are created and copied into their preinstallation locations. If the build was successful, you should create a test script. Open the test script template `test.pl` and add the following lines to the bottom:

```
@shuffle = shuffle 1..10;
print "@shuffle\n";
print "ok 2\n";
```

Now, type:

```
% make test
```

You should see something like the following at the bottom of the output:

```
ok 1
6 4 1 5 3 7 10 2 8 9
ok 2
```

Voila! Now you just need to write the documentation POD (see Item 46). This is left as an exercise to the reader.

Let's take this module and run some benchmarks. In the process we will also try out the `blib` pragma (see Item 43). Create a short benchmark program, called `tryme` as usual:

```
use Benchmark;
use List::Shuffle;
# insert shuffle1() and shuffle2() from above here
timethese(500, {
    'shuffle' => 'shuffle 1..1000',
    'shuffle1' => 'shuffle1 1..1000',
    'shuffle2' => 'shuffle2 1..1000',
});
```

Even though `List::Shuffle` isn't installed, we can use it if we tell Perl how to find the copy in the build library. The usual way is to invoke the `blib` pragma from the command line (this assumes `tryme` is in the same directory as the `blib` directory):

```
% perl -Mblib tryme
Using /home/joseph/perl-play/xs/List/Shuffle/blib
Benchmark: timing 500 iterations of shuffle, shuffle1,
shuffle2...
  shuffle:  3 secs ( 3.31 usr  0.00 sys =  3.31 cpu)
  shuffle1: 28 secs (28.79 usr  0.00 sys = 28.79 cpu)
  shuffle2: 31 secs (30.68 usr  0.00 sys = 30.68 cpu)
```

Well, that's an improvement in speed. As you can see, the XSUB shuffle ran approximately eight times faster (on my machine, anyway) than the two versions written in Perl. With better stack handling and some other optimizations it could be made a little faster still. But we will have to leave that as another exercise for the interested reader.

## Item 48: Submit your useful modules to the CPAN.

Although the CPAN already contains many high-quality modules that do a wide variety of things, there are still many useful modules yet to be written in Perl.

If you are considering writing a module that does something of general interest, or if you have already written one, you might want to consider contributing your module to the CPAN. The process of contributing a module is a fairly simple one—which is good, because we programmers hate it when paperwork becomes more time-consuming than the constructive work it supports.<sup>4</sup>

### Discuss your module

The first thing you need to do, as a potential new contributor to the CPAN, is to discuss your new module(s) with some potential users. In most cases, the best thing to do is to post some sort of a “request for discussion” to `comp.lang.perl.modules`. Include the proposed name of the module, its key features, and if possible, a pointer to a man page and/or a sample implementation. You don't have to take all the responses to heart, but you should at least read and consider them. Some people are just naturally “bah humbuggers” while others are helpful and interested—USENET has a lot of both kinds.

The following questions are ones that you need to have answered when you proceed:

---

4. This Item relies on the PAUSE documentation available at the time of writing. The current version can be found under `modules/04pause.html` in the CPAN. Please check it before registering or contributing modules for the first time, as the procedure may have changed since this was written.