NAME

perlcall - Perl calling conventions from C

DESCRIPTION

The purpose of this document is to show you how to call Perl subroutines directly from C, i.e., how to write *callbacks*.

Apart from discussing the C interface provided by Perl for writing callbacks the document uses a series of examples to show how the interface actually works in practice. In addition some techniques for coding callbacks are covered.

Examples where callbacks are necessary include

* An Error Handler

  You have created an XSUB interface to an application's C API.

  A fairly common feature in applications is to allow you to define a C function that will be called whenever something nasty occurs. What we would like is to be able to specify a Perl subroutine that will be called instead.

* An Event Driven Program

  The classic example of where callbacks are used is when writing an event driven program like for an X windows application. In this case you register functions to be called whenever specific events occur, e.g., a mouse button is pressed, the cursor moves into a window or a menu item is selected.

Although the techniques described here are applicable when embedding Perl in a C program, this is not the primary goal of this document. There are other details that must be considered and are specific to embedding Perl. For details on embedding Perl in C refer to *perlembed*.

Before you launch yourself head first into the rest of this document, it would be a good idea to have read the following two documents - *perlxs* and *perlguts*.

THE CALL_ FUNCTIONS

Although this stuff is easier to explain using examples, you first need be aware of a few important definitions.

Perl has a number of C functions that allow you to call Perl subroutines. They are

```
I32 call_sv(SV* sv, I32 flags);
I32 call_pv(char *subname, I32 flags);
I32 call_method(char *methname, I32 flags);
I32 call_argv(char *subname, I32 flags, register char **argv);
```

The key function is *call_sv*. All the other functions are fairly simple wrappers which make it easier to call Perl subroutines in special cases. At the end of the day they will all call *call_sv* to invoke the Perl subroutine.

All the *call_** functions have a `flags` parameter which is used to pass a bit mask of options to Perl. This bit mask operates identically for each of the functions. The settings available in the bit mask are discussed in *FLAG VALUES*.

Each of the functions will now be discussed in turn.

call_sv

call_sv takes two parameters, the first, `sv`, is an SV*. This allows you to specify the Perl subroutine to be called either as a C string (which has first been converted to an SV) or a reference to a subroutine. The section, *Using call_sv*, shows how you can make use of *call_sv*.

call_pv

The function, *call_pv*, is similar to *call_sv* except it expects its first parameter to be a C char* which identifies the Perl subroutine you want to call, e.g., `call_pv("fred", 0)`. If the subroutine you want to call is in another package, just include the package name in the string, e.g., `"pkg::fred"`.

call_method

The function *call_method* is used to call a method from a Perl class. The parameter `methname` corresponds to the name of the method to be called. Note that the class that the method belongs to is passed on the Perl stack rather than in the parameter list. This class can be either the name of the class (for a static method) or a reference to an object (for a virtual method). See *perlobj* for more information on static and virtual methods and *Using call_method* for an example of using *call_method*.

call_argv

call_argv calls the Perl subroutine specified by the C string stored in the `subname` parameter. It also takes the usual `flags` parameter. The final parameter, `argv`, consists of a NULL terminated list of C strings to be passed as parameters to the Perl subroutine. See *Using call_argv*.

All the functions return an integer. This is a count of the number of items returned by the Perl subroutine. The actual items returned by the subroutine are stored on the Perl stack.

As a general rule you should *always* check the return value from these functions. Even if you are expecting only a particular number of values to be returned from the Perl subroutine, there is nothing to stop someone from doing something unexpected--don't say you haven't been warned.

FLAG VALUES

The `flags` parameter in all the *call_** functions is a bit mask which can consist of any combination of the symbols defined below, OR'ed together.

G_VOID

Calls the Perl subroutine in a void context.

This flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a void context (if it executes *wantarray* the result will be the undefined value).

2. It ensures that nothing is actually returned from the subroutine.

The value returned by the *call_** function indicates how many items have been returned by the Perl subroutine - in this case it will be 0.

G_SCALAR

Calls the Perl subroutine in a scalar context. This is the default context flag setting for all the *call_** functions.

This flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a scalar context (if it executes *wantarray* the result will be false).

2. It ensures that only a scalar is actually returned from the subroutine. The subroutine can, of course, ignore the *wantarray* and return a list anyway. If so, then only the last element of the list will be returned.

The value returned by the *call_** function indicates how many items have been returned by the Perl subroutine - in this case it will be either 0 or 1.

If 0, then you have specified the G_DISCARD flag.

If 1, then the item actually returned by the Perl subroutine will be stored on the Perl stack - the section *Returning a Scalar* shows how to access this value on the stack. Remember that regardless of how many items the Perl subroutine returns, only the last one will be accessible from the stack - think of the case where only one value is returned as being a list with only one element. Any other items that were returned will not exist by the time control returns from the *call_** function. The section *Returning a list in a scalar context* shows an example of this behavior.

G_ARRAY

Calls the Perl subroutine in a list context.

As with G_SCALAR, this flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a list context (if it executes *wantarray* the result will be true).

2. It ensures that all items returned from the subroutine will be accessible when control returns from the *call_** function.

The value returned by the *call_** function indicates how many items have been returned by the Perl subroutine.

If 0, then you have specified the G_DISCARD flag.

If not 0, then it will be a count of the number of items returned by the subroutine. These items will be stored on the Perl stack. The section *Returning a list of values* gives an example of using the G_ARRAY flag and the mechanics of accessing the returned items from the Perl stack.

G_DISCARD

By default, the *call_** functions place the items returned from by the Perl subroutine on the stack. If you are not interested in these items, then setting this flag will make Perl get rid of them automatically for you. Note that it is still possible to indicate a context to the Perl subroutine by using either G_SCALAR or G_ARRAY.

If you do not set this flag then it is *very* important that you make sure that any temporaries (i.e., parameters passed to the Perl subroutine and values returned from the subroutine) are disposed of yourself. The section *Returning a Scalar* gives details of how to dispose of these temporaries explicitly and the section *Using Perl to dispose of temporaries* discusses the specific circumstances where you can ignore the problem and let Perl deal with it for you.

G_NOARGS

Whenever a Perl subroutine is called using one of the *call_** functions, it is assumed by default that parameters are to be passed to the subroutine. If you are not passing any parameters to the Perl subroutine, you can save a bit of time by setting this flag. It has the effect of not creating the @_ array for the Perl subroutine.

Although the functionality provided by this flag may seem straightforward, it should be used only if there is a good reason to do so. The reason for being cautious is that even if you have specified the G_NOARGS flag, it is still possible for the Perl subroutine that has been called to think that you have passed it parameters.

In fact, what can happen is that the Perl subroutine you have called can access the @_ array from a previous Perl subroutine. This will occur when the code that is executing the *call_** function has itself been called from another Perl subroutine. The code below illustrates this

```
sub fred
  { print "@_\n"  }
```

```
sub joe
  { &fred }

&joe(1,2,3);
```

This will print

```
1 2 3
```

What has happened is that `fred` accesses the `@_` array which belongs to `joe`.

G_EVAL

It is possible for the Perl subroutine you are calling to terminate abnormally, e.g., by calling *die* explicitly or by not actually existing. By default, when either of these events occurs, the process will terminate immediately. If you want to trap this type of event, specify the G_EVAL flag. It will put an *eval { }* around the subroutine call.

Whenever control returns from the *call_** function you need to check the $@ variable as you would in a normal Perl script.

The value returned from the *call_** function is dependent on what other flags have been specified and whether an error has occurred. Here are all the different cases that can occur:

- If the *call_** function returns normally, then the value returned is as specified in the previous sections.
- If G_DISCARD is specified, the return value will always be 0.
- If G_ARRAY is specified *and* an error has occurred, the return value will always be 0.
- If G_SCALAR is specified *and* an error has occurred, the return value will be 1 and the value on the top of the stack will be *undef*. This means that if you have already detected the error by checking $@ and you want the program to continue, you must remember to pop the *undef* from the stack.

See *Using G_EVAL* for details on using G_EVAL.

G_KEEPERR

You may have noticed that using the G_EVAL flag described above will **always** clear the $@ variable and set it to a string describing the error iff there was an error in the called code. This unqualified resetting of $@ can be problematic in the reliable identification of errors using the `eval {}` mechanism, because the possibility exists that perl will call other code (end of block processing code, for example) between the time the error causes $@ to be set within `eval {}`, and the subsequent statement which checks for the value of $@ gets executed in the user's script.

This scenario will mostly be applicable to code that is meant to be called from within destructors, asynchronous callbacks, signal handlers, `__DIE__` or `__WARN__` hooks, and `tie` functions. In such situations, you will not want to clear $@ at all, but simply to append any new errors to any existing value of $@.

The G_KEEPERR flag is meant to be used in conjunction with G_EVAL in *call_** functions that are used to implement such code. This flag has no effect when G_EVAL is not used.

When G_KEEPERR is used, any errors in the called code will be prefixed with the string "\t(in cleanup)", and appended to the current value of $@. an error will not be appended if that same error string is already at the end of $@.

In addition, a warning is generated using the appended string. This can be disabled using `no warnings 'misc'`.

The G_KEEPERR flag was introduced in Perl version 5.002.

See *Using G_KEEPERR* for an example of a situation that warrants the use of this flag.

Determining the Context

As mentioned above, you can determine the context of the currently executing subroutine in Perl with *wantarray*. The equivalent test can be made in C by using the `GIMME_V` macro, which returns `G_ARRAY` if you have been called in a list context, `G_SCALAR` if in a scalar context, or `G_VOID` if in a void context (i.e. the return value will not be used). An older version of this macro is called `GIMME`; in a void context it returns `G_SCALAR` instead of `G_VOID`. An example of using the `GIMME_V` macro is shown in section *Using GIMME_V*.

EXAMPLES

Enough of the definition talk, let's have a few examples.

Perl provides many macros to assist in accessing the Perl stack. Wherever possible, these macros should always be used when interfacing to Perl internals. We hope this should make the code less vulnerable to any changes made to Perl in the future.

Another point worth noting is that in the first series of examples I have made use of only the *call_pv* function. This has been done to keep the code simpler and ease you into the topic. Wherever possible, if the choice is between using *call_pv* and *call_sv*, you should always try to use *call_sv*. See *Using call_sv* for details.

No Parameters, Nothing returned

This first trivial example will call a Perl subroutine, *PrintUID*, to print out the UID of the process.

```
sub PrintUID
{
    print "UID is $<\n";
}
```

and here is a C function to call it

```
static void
call_PrintUID()
{
    dSP;

    PUSHMARK(SP);
    call_pv("PrintUID", G_DISCARD|G_NOARGS);
}
```

Simple, eh.

A few points to note about this example.

1. Ignore `dSP` and `PUSHMARK(SP)` for now. They will be discussed in the next example.
2. We aren't passing any parameters to *PrintUID* so G_NOARGS can be specified.
3. We aren't interested in anything returned from *PrintUID*, so G_DISCARD is specified. Even if *PrintUID* was changed to return some value(s), having specified G_DISCARD will mean that they will be wiped by the time control returns from *call_pv*.
4. As *call_pv* is being used, the Perl subroutine is specified as a C string. In this case the subroutine name has been 'hard-wired' into the code.
5. Because we specified G_DISCARD, it is not necessary to check the value returned from

call_pv. It will always be 0.

Passing Parameters

Now let's make a slightly more complex example. This time we want to call a Perl subroutine, `LeftString`, which will take 2 parameters--a string ($s) and an integer ($n). The subroutine will simply print the first $n characters of the string.

So the Perl subroutine would look like this

```
sub LeftString
{
    my($s, $n) = @_;
    print substr($s, 0, $n), "\n";
}
```

The C function required to call *LeftString* would look like this.

```
static void
call_LeftString(a, b)
char * a;
int b;
{
    dSP;

ENTER;
    SAVETMPS;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSVpv(a, 0)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK;

    call_pv("LeftString", G_DISCARD);

    FREETMPS;
    LEAVE;
}
```

Here are a few notes on the C function *call_LeftString*.

1. Parameters are passed to the Perl subroutine using the Perl stack. This is the purpose of the code beginning with the line `dSP` and ending with the line `PUTBACK`. The `dSP` declares a local copy of the stack pointer. This local copy should **always** be accessed as `SP`.

2. If you are going to put something onto the Perl stack, you need to know where to put it. This is the purpose of the macro dSP--it declares and initializes a *local* copy of the Perl stack pointer.

   All the other macros which will be used in this example require you to have used this macro.

   The exception to this rule is if you are calling a Perl subroutine directly from an XSUB function. In this case it is not necessary to use the `dSP` macro explicitly--it will be declared for you automatically.

3. Any parameters to be pushed onto the stack should be bracketed by the `PUSHMARK` and `PUTBACK` macros. The purpose of these two macros, in this context, is to count the number of parameters you are pushing automatically. Then whenever Perl is creating the `@_` array

for the subroutine, it knows how big to make it.

The PUSHMARK macro tells Perl to make a mental note of the current stack pointer. Even if you aren't passing any parameters (like the example shown in the section *No Parameters, Nothing returned*) you must still call the PUSHMARK macro before you can call any of the *call_** functions--Perl still needs to know that there are no parameters.

The PUTBACK macro sets the global copy of the stack pointer to be the same as our local copy. If we didn't do this *call_pv* wouldn't know where the two parameters we pushed were--remember that up to now all the stack pointer manipulation we have done is with our local copy, *not* the global copy.

4. Next, we come to XPUSHs. This is where the parameters actually get pushed onto the stack. In this case we are pushing a string and an integer.

   See *"XSUBs and the Argument Stack" in perlguts* for details on how the XPUSH macros work.

5. Because we created temporary values (by means of sv_2mortal() calls) we will have to tidy up the Perl stack and dispose of mortal SVs.

   This is the purpose of

   ```
   ENTER;
   SAVETMPS;
   ```

   at the start of the function, and

   ```
   FREETMPS;
   LEAVE;
   ```

   at the end. The ENTER/SAVETMPS pair creates a boundary for any temporaries we create. This means that the temporaries we get rid of will be limited to those which were created after these calls.

   The FREETMPS/LEAVE pair will get rid of any values returned by the Perl subroutine (see next example), plus it will also dump the mortal SVs we have created. Having ENTER/SAVETMPS at the beginning of the code makes sure that no other mortals are destroyed.

   Think of these macros as working a bit like using { and } in Perl to limit the scope of local variables.

   See the section *Using Perl to dispose of temporaries* for details of an alternative to using these macros.

6. Finally, *LeftString* can now be called via the *call_pv* function. The only flag specified this time is G_DISCARD. Because we are passing 2 parameters to the Perl subroutine this time, we have not specified G_NOARGS.

Returning a Scalar

Now for an example of dealing with the items returned from a Perl subroutine.

Here is a Perl subroutine, *Adder*, that takes 2 integer parameters and simply returns their sum.

```
sub Adder
{
    my($a, $b) = @_;
    $a + $b;
}
```

Because we are now concerned with the return value from *Adder*, the C function required to call it is now a bit more complex.

```
static void
```

```
call_Adder(a, b)
int a;
int b;
{
    dSP;
    int count;

    ENTER;
    SAVETMPS;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK;

    count = call_pv("Adder", G_SCALAR);

    SPAGAIN;

    if (count != 1)
        croak("Big trouble\n");

    printf ("The sum of %d and %d is %d\n", a, b, POPi);

    PUTBACK;
    FREETMPS;
    LEAVE;
}
```

Points to note this time are

1. The only flag specified this time was G_SCALAR. That means the `@_` array will be created and that the value returned by *Adder* will still exist after the call to *call_pv*.

2. The purpose of the macro `SPAGAIN` is to refresh the local copy of the stack pointer. This is necessary because it is possible that the memory allocated to the Perl stack has been reallocated whilst in the *call_pv* call.

   If you are making use of the Perl stack pointer in your code you must always refresh the local copy using SPAGAIN whenever you make use of the *call_** functions or any other Perl internal function.

3. Although only a single value was expected to be returned from *Adder*, it is still good practice to check the return code from *call_pv* anyway.

   Expecting a single value is not quite the same as knowing that there will be one. If someone modified *Adder* to return a list and we didn't check for that possibility and take appropriate action the Perl stack would end up in an inconsistent state. That is something you *really* don't want to happen ever.

4. The `POPi` macro is used here to pop the return value from the stack. In this case we wanted an integer, so `POPi` was used.

   Here is the complete list of POP macros available, along with the types they return.

   ```
   POPs SV
   POPp pointer
   ```

```
POPn double
POPi integer
POPl long
```

5. The final PUTBACK is used to leave the Perl stack in a consistent state before exiting the function. This is necessary because when we popped the return value from the stack with POPi it updated only our local copy of the stack pointer. Remember, PUTBACK sets the global stack pointer to be the same as our local copy.

Returning a list of values

Now, let's extend the previous example to return both the sum of the parameters and the difference.

Here is the Perl subroutine

```
sub AddSubtract
{
   my($a, $b) = @_;
   ($a+$b, $a-$b);
}
```

and this is the C function

```
static void
call_AddSubtract(a, b)
int a;
int b;
{
    dSP;
    int count;

    ENTER;
    SAVETMPS;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK;

    count = call_pv("AddSubtract", G_ARRAY);

    SPAGAIN;

    if (count != 2)
        croak("Big trouble\n");

    printf ("%d - %d = %d\n", a, b, POPi);
    printf ("%d + %d = %d\n", a, b, POPi);

    PUTBACK;
    FREETMPS;
    LEAVE;
}
```

If *call_AddSubtract* is called like this

```
call_AddSubtract(7, 4);
```

then here is the output

```
7 - 4 = 3
7 + 4 = 11
```

Notes

1. We wanted list context, so G_ARRAY was used.
2. Not surprisingly `POPi` is used twice this time because we were retrieving 2 values from the stack. The important thing to note is that when using the `POP*` macros they come off the stack in *reverse* order.

Returning a list in a scalar context

Say the Perl subroutine in the previous section was called in a scalar context, like this

```
static void
call_AddSubScalar(a, b)
int a;
int b;
{
    dSP;
    int count;
    int i;

    ENTER;
    SAVETMPS;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK;

    count = call_pv("AddSubtract", G_SCALAR);

    SPAGAIN;

    printf ("Items Returned = %d\n", count);

    for (i = 1; i <= count; ++i)
        printf ("Value %d = %d\n", i, POPi);

    PUTBACK;
    FREETMPS;
    LEAVE;
}
```

The other modification made is that *call_AddSubScalar* will print the number of items returned from the Perl subroutine and their value (for simplicity it assumes that they are integer). So if *call_AddSubScalar* is called

```
call_AddSubScalar(7, 4);
```

then the output will be

```
Items Returned = 1
Value 1 = 3
```

In this case the main point to note is that only the last item in the list is returned from the subroutine, *AddSubtract* actually made it back to *call_AddSubScalar*.

Returning Data from Perl via the parameter list

It is also possible to return values directly via the parameter list - whether it is actually desirable to do it is another matter entirely.

The Perl subroutine, *Inc*, below takes 2 parameters and increments each directly.

```
sub Inc
{
    ++ $_[0];
    ++ $_[1];
}
```

and here is a C function to call it.

```
static void
call_Inc(a, b)
int a;
int b;
{
    dSP;
    int count;
    SV * sva;
    SV * svb;

    ENTER;
    SAVETMPS;

    sva = sv_2mortal(newSViv(a));
    svb = sv_2mortal(newSViv(b));

    PUSHMARK(SP);
    XPUSHs(sva);
    XPUSHs(svb);
    PUTBACK;

    count = call_pv("Inc", G_DISCARD);

    if (count != 0)
        croak ("call_Inc: expected 0 values from 'Inc', got %d\n",
               count);

    printf ("%d + 1 = %d\n", a, SvIV(sva));
    printf ("%d + 1 = %d\n", b, SvIV(svb));

FREETMPS;
LEAVE;
```

```
}
```

To be able to access the two parameters that were pushed onto the stack after they return from *call_pv* it is necessary to make a note of their addresses--thus the two variables `sva` and `svb`.

The reason this is necessary is that the area of the Perl stack which held them will very likely have been overwritten by something else by the time control returns from *call_pv*.

Using G_EVAL

Now an example using G_EVAL. Below is a Perl subroutine which computes the difference of its 2 parameters. If this would result in a negative result, the subroutine calls *die*.

```
sub Subtract
{
    my ($a, $b) = @_;

    die "death can be fatal\n" if $a < $b;

    $a - $b;
}
```

and some C to call it

```
static void
call_Subtract(a, b)
int a;
int b;
{
    dSP;
    int count;

    ENTER;
    SAVETMPS;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK;

    count = call_pv("Subtract", G_EVAL|G_SCALAR);

    SPAGAIN;

    /* Check the eval first */
    if (SvTRUE(ERRSV))
    {
 STRLEN n_a;
        printf ("Uh oh - %s\n", SvPV(ERRSV, n_a));
        POPs;
    }
    else
    {
        if (count != 1)
           croak("call_Subtract: wanted 1 value from 'Subtract', got
```

```
%d\n",                          count);

            printf ("%d - %d = %d\n", a, b, POPi);
        }

        PUTBACK;
        FREETMPS;
        LEAVE;
    }
```

If *call_Subtract* is called thus

```
call_Subtract(4, 5)
```

the following will be printed

```
Uh oh - death can be fatal
```

Notes

1. We want to be able to catch the *die* so we have used the G_EVAL flag. Not specifying this flag would mean that the program would terminate immediately at the *die* statement in the subroutine *Subtract*.

2. The code

   ```
       if (SvTRUE(ERRSV))
       {
    STRLEN n_a;
           printf ("Uh oh - %s\n", SvPV(ERRSV, n_a));
           POPs;
       }
   ```

   is the direct equivalent of this bit of Perl

   ```
       print "Uh oh - $@\n" if $@;
   ```

   `PL_errgv` is a perl global of type `GV *` that points to the symbol table entry containing the error. ERRSV therefore refers to the C equivalent of `$@`.

3. Note that the stack is popped using `POPs` in the block where `SvTRUE(ERRSV)` is true. This is necessary because whenever a *call_** function invoked with G_EVAL|G_SCALAR returns an error, the top of the stack holds the value *undef*. Because we want the program to continue after detecting this error, it is essential that the stack is tidied up by removing the *undef*.

Using G_KEEPERR

Consider this rather facetious example, where we have used an XS version of the call_Subtract example above inside a destructor:

```
package Foo;
sub new { bless {}, $_[0] }
sub Subtract {
    my($a,$b) = @_;
    die "death can be fatal" if $a < $b;
    $a - $b;
}
sub DESTROY { call_Subtract(5, 4); }
```

```
    sub foo { die "foo dies"; }

    package main;
    eval { Foo->new->foo };
    print "Saw: $@" if $@;             # should be, but isn't
```

This example will fail to recognize that an error occurred inside the `eval {}`. Here's why: the call_Subtract code got executed while perl was cleaning up temporaries when exiting the eval block, and because call_Subtract is implemented with *call_pv* using the G_EVAL flag, it promptly reset $@. This results in the failure of the outermost test for $@, and thereby the failure of the error trap.

Appending the G_KEEPERR flag, so that the *call_pv* call in call_Subtract reads:

```
        count = call_pv("Subtract", G_EVAL|G_SCALAR|G_KEEPERR);
```

will preserve the error and restore reliable error handling.

Using call_sv

In all the previous examples I have 'hard-wired' the name of the Perl subroutine to be called from C. Most of the time though, it is more convenient to be able to specify the name of the Perl subroutine from within the Perl script.

Consider the Perl code below

```
    sub fred
    {
        print "Hello there\n";
    }

    CallSubPV("fred");
```

Here is a snippet of XSUB which defines *CallSubPV*.

```
    void
    CallSubPV(name)
     char * name
     CODE:
 PUSHMARK(SP);
 call_pv(name, G_DISCARD|G_NOARGS);
```

That is fine as far as it goes. The thing is, the Perl subroutine can be specified as only a string. For Perl 4 this was adequate, but Perl 5 allows references to subroutines and anonymous subroutines. This is where *call_sv* is useful.

The code below for *CallSubSV* is identical to *CallSubPV* except that the `name` parameter is now defined as an SV* and we use *call_sv* instead of *call_pv*.

```
    void
    CallSubSV(name)
     SV * name
     CODE:
 PUSHMARK(SP);
 call_sv(name, G_DISCARD|G_NOARGS);
```

Because we are using an SV to call *fred* the following can all be used

```
    CallSubSV("fred");
    CallSubSV(\&fred);
    $ref = \&fred;
    CallSubSV($ref);
    CallSubSV( sub { print "Hello there\n" } );
```

As you can see, *call_sv* gives you much greater flexibility in how you can specify the Perl subroutine.

You should note that if it is necessary to store the SV (`name` in the example above) which corresponds to the Perl subroutine so that it can be used later in the program, it not enough just to store a copy of the pointer to the SV. Say the code above had been like this

```
    static SV * rememberSub;

    void
    SaveSub1(name)
     SV * name
     CODE:
 rememberSub = name;

    void
    CallSavedSub1()
     CODE:
 PUSHMARK(SP);
 call_sv(rememberSub, G_DISCARD|G_NOARGS);
```

The reason this is wrong is that by the time you come to use the pointer `rememberSub` in `CallSavedSub1`, it may or may not still refer to the Perl subroutine that was recorded in `SaveSub1`. This is particularly true for these cases

```
    SaveSub1(\&fred);
    CallSavedSub1();

    SaveSub1( sub { print "Hello there\n" } );
    CallSavedSub1();
```

By the time each of the `SaveSub1` statements above have been executed, the SV*s which corresponded to the parameters will no longer exist. Expect an error message from Perl of the form

```
    Can't use an undefined value as a subroutine reference at ...
```

for each of the `CallSavedSub1` lines.

Similarly, with this code

```
    $ref = \&fred;
    SaveSub1($ref);
    $ref = 47;
    CallSavedSub1();
```

you can expect one of these messages (which you actually get is dependent on the version of Perl you are using)

```
    Not a CODE reference at ...
    Undefined subroutine &main::47 called ...
```

The variable $ref may have referred to the subroutine `fred` whenever the call to `SaveSub1` was made but by the time `CallSavedSub1` gets called it now holds the number `47`. Because we saved only a pointer to the original SV in `SaveSub1`, any changes to $ref will be tracked by the pointer `rememberSub`. This means that whenever `CallSavedSub1` gets called, it will attempt to execute the code which is referenced by the SV* `rememberSub`. In this case though, it now refers to the integer `47`, so expect Perl to complain loudly.

A similar but more subtle problem is illustrated with this code

```
$ref = \&fred;
SaveSub1($ref);
$ref = \&joe;
CallSavedSub1();
```

This time whenever `CallSavedSub1` get called it will execute the Perl subroutine `joe` (assuming it exists) rather than `fred` as was originally requested in the call to `SaveSub1`.

To get around these problems it is necessary to take a full copy of the SV. The code below shows `SaveSub2` modified to do that

```
static SV * keepSub = (SV*)NULL;

void
SaveSub2(name)
    SV * name
 CODE:
  /* Take a copy of the callback */
 if (keepSub == (SV*)NULL)
     /* First time, so create a new SV */
 keepSub = newSVsv(name);
 else
     /* Been here before, so overwrite */
 SvSetSV(keepSub, name);

void
CallSavedSub2()
 CODE:
PUSHMARK(SP);
call_sv(keepSub, G_DISCARD|G_NOARGS);
```

To avoid creating a new SV every time `SaveSub2` is called, the function first checks to see if it has been called before. If not, then space for a new SV is allocated and the reference to the Perl subroutine, `name` is copied to the variable `keepSub` in one operation using `newSVsv`. Thereafter, whenever `SaveSub2` is called the existing SV, `keepSub`, is overwritten with the new value using `SvSetSV`.

Using call_argv

Here is a Perl subroutine which prints whatever parameters are passed to it.

```
sub PrintList
{
    my(@list) = @_;

    foreach (@list) { print "$_\n" }
}
```

and here is an example of *call_argv* which will call *PrintList*.

```
static char * words[] = {"alpha", "beta", "gamma", "delta", NULL};

static void
call_PrintList()
{
    dSP;

    call_argv("PrintList", G_DISCARD, words);
}
```

Note that it is not necessary to call PUSHMARK in this instance. This is because *call_argv* will do it for you.

Using call_method

Consider the following Perl code

```
{
    package Mine;

    sub new
    {
        my($type) = shift;
        bless [@_]
    }

    sub Display
    {
        my ($self, $index) = @_;
        print "$index: $$self[$index]\n";
    }

    sub PrintID
    {
        my($class) = @_;
        print "This is Class $class version 1.0\n";
    }
}
```

It implements just a very simple class to manage an array. Apart from the constructor, `new`, it declares methods, one static and one virtual. The static method, `PrintID`, prints out simply the class name and a version number. The virtual method, `Display`, prints out a single element of the array. Here is an all Perl example of using it.

```
$a = new Mine ('red', 'green', 'blue');
$a->Display(1);
PrintID Mine;
```

will print

```
1: green
This is Class Mine version 1.0
```

Calling a Perl method from C is fairly straightforward. The following things are required

- a reference to the object for a virtual method or the name of the class for a static method.
- the name of the method.
- any other parameters specific to the method.

Here is a simple XSUB which illustrates the mechanics of calling both the `PrintID` and `Display` methods from C.

```
void
call_Method(ref, method, index)
    SV * ref
    char * method
    int  index
    CODE:
    PUSHMARK(SP);
    XPUSHs(ref);
    XPUSHs(sv_2mortal(newSViv(index)));
    PUTBACK;

    call_method(method, G_DISCARD);

void
call_PrintID(class, method)
    char * class
    char * method
    CODE:
    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSVpv(class, 0)));
    PUTBACK;

    call_method(method, G_DISCARD);
```

So the methods `PrintID` and `Display` can be invoked like this

```
$a = new Mine ('red', 'green', 'blue');
call_Method($a, 'Display', 1);
call_PrintID('Mine', 'PrintID');
```

The only thing to note is that in both the static and virtual methods, the method name is not passed via the stack--it is used as the first parameter to *call_method*.

Using GIMME_V

Here is a trivial XSUB which prints the context in which it is currently executing.

```
void
PrintContext()
    CODE:
    I32 gimme = GIMME_V;
    if (gimme == G_VOID)
        printf ("Context is Void\n");
    else if (gimme == G_SCALAR)
        printf ("Context is Scalar\n");
    else
        printf ("Context is Array\n");
```

and here is some Perl to test it

```
PrintContext;
$a = PrintContext;
@a = PrintContext;
```

The output from that will be

```
Context is Void
Context is Scalar
Context is Array
```

Using Perl to dispose of temporaries

In the examples given to date, any temporaries created in the callback (i.e., parameters passed on the stack to the *call_** function or values returned via the stack) have been freed by one of these methods

- specifying the G_DISCARD flag with *call_**.
- explicitly disposed of using the ENTER/SAVETMPS - FREETMPS/LEAVE pairing.

There is another method which can be used, namely letting Perl do it for you automatically whenever it regains control after the callback has terminated. This is done by simply not using the

```
ENTER;
SAVETMPS;
...
FREETMPS;
LEAVE;
```

sequence in the callback (and not, of course, specifying the G_DISCARD flag).

If you are going to use this method you have to be aware of a possible memory leak which can arise under very specific circumstances. To explain these circumstances you need to know a bit about the flow of control between Perl and the callback routine.

The examples given at the start of the document (an error handler and an event driven program) are typical of the two main sorts of flow control that you are likely to encounter with callbacks. There is a very important distinction between them, so pay attention.

In the first example, an error handler, the flow of control could be as follows. You have created an interface to an external library. Control can reach the external library like this

```
perl --> XSUB --> external library
```

Whilst control is in the library, an error condition occurs. You have previously set up a Perl callback to handle this situation, so it will get executed. Once the callback has finished, control will drop back to Perl again. Here is what the flow of control will be like in that situation

```
perl --> XSUB --> external library
                  ...
                  error occurs
                  ...
                  external library --> call_* --> perl
                                                    |
perl <-- XSUB <-- external library <-- call_* <----+
```

After processing of the error using *call_** is completed, control reverts back to Perl more or less

immediately.In the diagram, the further right you go the more deeply nested the scope is. It is only when control is back with perl on the extreme left of the diagram that you will have dropped back to the enclosing scope and any temporaries you have left hanging around will be freed.

In the second example, an event driven program, the flow of control will be more like this

```
perl --> XSUB --> event handler
                  ...
                  event handler --> call_* --> perl
                                                 |
                  event handler <-- call_* <----+
                  ...
                  event handler --> call_* --> perl
                                                 |
                  event handler <-- call_* <----+
                  ...
                  event handler --> call_* --> perl
                                                 |
                  event handler <-- call_* <----+
```

In this case the flow of control can consist of only the repeated sequence

```
event handler --> call_* --> perl
```

for practically the complete duration of the program. This means that control may *never* drop back to the surrounding scope in Perl at the extreme left.

So what is the big problem? Well, if you are expecting Perl to tidy up those temporaries for you, you might be in for a long wait. For Perl to dispose of your temporaries, control must drop back to the enclosing scope at some stage. In the event driven scenario that may never happen. This means that as time goes on, your program will create more and more temporaries, none of which will ever be freed. As each of these temporaries consumes some memory your program will eventually consume all the available memory in your system--kapow!

So here is the bottom line--if you are sure that control will revert back to the enclosing Perl scope fairly quickly after the end of your callback, then it isn't absolutely necessary to dispose explicitly of any temporaries you may have created. Mind you, if you are at all uncertain about what to do, it doesn't do any harm to tidy up anyway.

Strategies for storing Callback Context Information

Potentially one of the trickiest problems to overcome when designing a callback interface can be figuring out how to store the mapping between the C callback function and the Perl equivalent.

To help understand why this can be a real problem first consider how a callback is set up in an all C environment. Typically a C API will provide a function to register a callback. This will expect a pointer to a function as one of its parameters. Below is a call to a hypothetical function `register_fatal` which registers the C function to get called when a fatal error occurs.

```
register_fatal(cb1);
```

The single parameter `cb1` is a pointer to a function, so you must have defined `cb1` in your code, say something like this

```
static void
cb1()
{
    printf ("Fatal Error\n");
    exit(1);
```

```
}
```

Now change that to call a Perl subroutine instead

```
static SV * callback = (SV*)NULL;

static void
cb1()
{
    dSP;

    PUSHMARK(SP);

    /* Call the Perl sub to process the callback */
    call_sv(callback, G_DISCARD);
}

void
register_fatal(fn)
    SV * fn
    CODE:
    /* Remember the Perl sub */
    if (callback == (SV*)NULL)
        callback = newSVsv(fn);
    else
        SvSetSV(callback, fn);

    /* register the callback with the external library */
    register_fatal(cb1);
```

where the Perl equivalent of `register_fatal` and the callback it registers, `pcb1`, might look like this

```
# Register the sub pcb1
register_fatal(\&pcb1);

sub pcb1
{
    die "I'm dying...\n";
}
```

The mapping between the C callback and the Perl equivalent is stored in the global variable `callback`.

This will be adequate if you ever need to have only one callback registered at any time. An example could be an error handler like the code sketched out above. Remember though, repeated calls to `register_fatal` will replace the previously registered callback function with the new one.

Say for example you want to interface to a library which allows asynchronous file i/o. In this case you may be able to register a callback whenever a read operation has completed. To be of any use we want to be able to call separate Perl subroutines for each file that is opened. As it stands, the error handler example above would not be adequate as it allows only a single callback to be defined at any time. What we require is a means of storing the mapping between the opened file and the Perl subroutine we want to be called for that file.

Say the i/o library has a function `asynch_read` which associates a C function `ProcessRead` with a file handle `fh`--this assumes that it has also provided some routine to open the file and so obtain the file handle.

```
asynch_read(fh, ProcessRead)
```

This may expect the C *ProcessRead* function of this form

```
void
ProcessRead(fh, buffer)
int fh;
char * buffer;
{
     ...
}
```

To provide a Perl interface to this library we need to be able to map between the `fh` parameter and the Perl subroutine we want called. A hash is a convenient mechanism for storing this mapping. The code below shows a possible implementation

```
static HV * Mapping = (HV*)NULL;

void
asynch_read(fh, callback)
    int fh
    SV * callback
    CODE:
    /* If the hash doesn't already exist, create it */
    if (Mapping == (HV*)NULL)
        Mapping = newHV();

    /* Save the fh -> callback mapping */
    hv_store(Mapping, (char*)&fh, sizeof(fh), newSVsv(callback), 0);

    /* Register with the C Library */
    asynch_read(fh, asynch_read_if);
```

and `asynch_read_if` could look like this

```
static void
asynch_read_if(fh, buffer)
int fh;
char * buffer;
{
    dSP;
    SV ** sv;

    /* Get the callback associated with fh */
    sv =  hv_fetch(Mapping, (char*)&fh , sizeof(fh), FALSE);
    if (sv == (SV**)NULL)
        croak("Internal error...\n");

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(fh)));
    XPUSHs(sv_2mortal(newSVpv(buffer, 0)));
```

```
        PUTBACK;

        /* Call the Perl sub */
        call_sv(*sv, G_DISCARD);
    }
```

For completeness, here is `asynch_close`. This shows how to remove the entry from the hash `Mapping`.

```
    void
    asynch_close(fh)
        int fh
        CODE:
        /* Remove the entry from the hash */
        (void) hv_delete(Mapping, (char*)&fh, sizeof(fh), G_DISCARD);

        /* Now call the real asynch_close */
        asynch_close(fh);
```

So the Perl interface would look like this

```
    sub callback1
    {
        my($handle, $buffer) = @_;
    }

    # Register the Perl callback
    asynch_read($fh, \&callback1);

    asynch_close($fh);
```

The mapping between the C callback and Perl is stored in the global hash `Mapping` this time. Using a hash has the distinct advantage that it allows an unlimited number of callbacks to be registered.

What if the interface provided by the C callback doesn't contain a parameter which allows the file handle to Perl subroutine mapping? Say in the asynchronous i/o package, the callback function gets passed only the `buffer` parameter like this

```
    void
    ProcessRead(buffer)
    char * buffer;
    {
        ...
    }
```

Without the file handle there is no straightforward way to map from the C callback to the Perl subroutine.

In this case a possible way around this problem is to predefine a series of C functions to act as the interface to Perl, thus

```
    #define MAX_CB  3
    #define NULL_HANDLE -1
    typedef void (*FnMap)();
```

```
struct MapStruct {
    FnMap    Function;
    SV *     PerlSub;
    int      Handle;
  };

static void  fn1();
static void  fn2();
static void  fn3();

static struct MapStruct Map [MAX_CB] =
    {
        { fn1, NULL, NULL_HANDLE },
        { fn2, NULL, NULL_HANDLE },
        { fn3, NULL, NULL_HANDLE }
    };

static void
Pcb(index, buffer)
int index;
char * buffer;
{
    dSP;

    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSVpv(buffer, 0)));
    PUTBACK;

    /* Call the Perl sub */
    call_sv(Map[index].PerlSub, G_DISCARD);
}

static void
fn1(buffer)
char * buffer;
{
    Pcb(0, buffer);
}

static void
fn2(buffer)
char * buffer;
{
    Pcb(1, buffer);
}

static void
fn3(buffer)
char * buffer;
{
    Pcb(2, buffer);
}
```

```
        void
        array_asynch_read(fh, callback)
            int  fh
            SV * callback
            CODE:
            int index;
            int null_index = MAX_CB;

            /* Find the same handle or an empty entry */
            for (index = 0; index < MAX_CB; ++index)
            {
                if (Map[index].Handle == fh)
                    break;

                if (Map[index].Handle == NULL_HANDLE)
                    null_index = index;
            }

            if (index == MAX_CB && null_index == MAX_CB)
                croak ("Too many callback functions registered\n");

            if (index == MAX_CB)
                index = null_index;

            /* Save the file handle */
            Map[index].Handle = fh;

            /* Remember the Perl sub */
            if (Map[index].PerlSub == (SV*)NULL)
                Map[index].PerlSub = newSVsv(callback);
            else
                SvSetSV(Map[index].PerlSub, callback);

            asynch_read(fh, Map[index].Function);

        void
        array_asynch_close(fh)
            int fh
            CODE:
            int index;

            /* Find the file handle */
            for (index = 0; index < MAX_CB; ++ index)
                if (Map[index].Handle == fh)
                    break;

            if (index == MAX_CB)
                croak ("could not close fh %d\n", fh);

            Map[index].Handle = NULL_HANDLE;
            SvREFCNT_dec(Map[index].PerlSub);
            Map[index].PerlSub = (SV*)NULL;
```

```
        asynch_close(fh);
```

In this case the functions `fn1`, `fn2`, and `fn3` are used to remember the Perl subroutine to be called. Each of the functions holds a separate hard-wired index which is used in the function `Pcb` to access the `Map` array and actually call the Perl subroutine.

There are some obvious disadvantages with this technique.

Firstly, the code is considerably more complex than with the previous example.

Secondly, there is a hard-wired limit (in this case 3) to the number of callbacks that can exist simultaneously. The only way to increase the limit is by modifying the code to add more functions and then recompiling. None the less, as long as the number of functions is chosen with some care, it is still a workable solution and in some cases is the only one available.

To summarize, here are a number of possible methods for you to consider for storing the mapping between C and the Perl callback

1. Ignore the problem - Allow only 1 callback

   For a lot of situations, like interfacing to an error handler, this may be a perfectly adequate solution.

2. Create a sequence of callbacks - hard wired limit

   If it is impossible to tell from the parameters passed back from the C callback what the context is, then you may need to create a sequence of C callback interface functions, and store pointers to each in an array.

3. Use a parameter to map to the Perl callback

   A hash is an ideal mechanism to store the mapping between C and Perl.

Alternate Stack Manipulation

Although I have made use of only the `POP*` macros to access values returned from Perl subroutines, it is also possible to bypass these macros and read the stack using the `ST` macro (See *perlxs* for a full description of the `ST` macro).

Most of the time the `POP*` macros should be adequate, the main problem with them is that they force you to process the returned values in sequence. This may not be the most suitable way to process the values in some cases. What we want is to be able to access the stack in a random order. The `ST` macro as used when coding an XSUB is ideal for this purpose.

The code below is the example given in the section *Returning a list of values* recoded to use `ST` instead of `POP*`.

```
    static void
    call_AddSubtract2(a, b)
    int a;
    int b;
    {
        dSP;
        I32 ax;
        int count;

        ENTER;
        SAVETMPS;

        PUSHMARK(SP);
        XPUSHs(sv_2mortal(newSViv(a)));
```

```
        XPUSHs(sv_2mortal(newSViv(b)));
        PUTBACK;

        count = call_pv("AddSubtract", G_ARRAY);

        SPAGAIN;
        SP -= count;
        ax = (SP - PL_stack_base) + 1;

        if (count != 2)
            croak("Big trouble\n");

        printf ("%d + %d = %d\n", a, b, SvIV(ST(0)));
        printf ("%d - %d = %d\n", a, b, SvIV(ST(1)));

        PUTBACK;
        FREETMPS;
        LEAVE;
    }
```

Notes

1. Notice that it was necessary to define the variable `ax`. This is because the `ST` macro expects it to exist. If we were in an XSUB it would not be necessary to define `ax` as it is already defined for you.

2. The code

   ```
   SPAGAIN;
   SP -= count;
   ax = (SP - PL_stack_base) + 1;
   ```

   sets the stack up so that we can use the `ST` macro.

3. Unlike the original coding of this example, the returned values are not accessed in reverse order. So `ST(0)` refers to the first value returned by the Perl subroutine and `ST(count-1)` refers to the last.

Creating and calling an anonymous subroutine in C

As we've already shown, `call_sv` can be used to invoke an anonymous subroutine. However, our example showed a Perl script invoking an XSUB to perform this operation. Let's see how it can be done inside our C code:

```
 ...

 SV *cvrv = eval_pv("sub { print 'You will not find me cluttering any
namespace!' }", TRUE);

 ...

 call_sv(cvrv, G_VOID|G_NOARGS);
```

`eval_pv` is used to compile the anonymous subroutine, which will be the return value as well (read more about `eval_pv` in *"eval_pv" in perlapi*). Once this code reference is in hand, it can be mixed in with all the previous examples we've shown.

SEE ALSO

perlxs, *perlguts*, *perlembed*

AUTHOR

Paul Marquess

Special thanks to the following people who assisted in the creation of the document.

Jeff Okamoto, Tim Bunce, Nick Gianniotis, Steve Kelem, Gurusamy Sarathy and Larry Wall.

DATE

Version 1.3, 14th Apr 1997