1 mod_perl 2.0 Source Code Explained

1.1 Description

This document explains how to navigate the mod_perl source code, modify and rebuild the existing code and most important: how to add new functionality.

1.2 Project's Filesystem Layout

In its pristine state the project is comprised of the following directories and files residing at the root directory of the project:

```
Apache-Test/      - test kit for mod_perl and Apache::* modules
ModPerl-Registry/ - ModPerl::Registry sub-project
build/            - utilities used during project build
docs/             - documentation
lib/              - Perl modules
src/              - C code that builds libmodperl.so
t/                - mod_perl tests
todo/             - things to be done
util/             - useful utilities for developers
xs/               - source xs code and maps
Changes           - Changes file
LICENSE           - ASF LICENSE document
Makefile.PL       - generates all the needed Makefiles
```

After building the project, the following root directories and files get generated:

```
Makefile     - Makefile
WrapXS/      - autogenerated XS code
blib/        - ready to install version of the package
```

1.3 Directory src

1.3.1 Directory src/modules/perl/

The directory *src/modules/perl* includes the C source files needed to build the *libmodperl* library.

Notice that several files in this directory are autogenerated during the *perl Makefile* stage.

When adding new source files to this directory you should add their names to the `@c_src_names` variable in *lib/ModPerl/Code.pm*, so they will be picked up by the autogenerated *Makefile*.

1.4 Directory xs/

```
Apache/                - Apache specific XS code
APR/                   - APR specific XS code
ModPerl/               - ModPerl specific XS code
maps/                  -
tables/                -
Makefile.PL            -
```

```
modperl_xs_sv_convert.h -
modperl_xs_typedefs.h   -
modperl_xs_util.h       -
typemap                 -
```

1.4.1 xs/Apache, xs/APR and xs/ModPerl

The *xs/Apache*, *xs/APR* and *xs/ModPerl* directories include *.h* files which have C and XS code in them. They all have the *.h* extension because they are always `#include-d`, never compiled into their own object file. and only the file that `#include-s` an *.h* file from these directories should be able to see what's in there. Anything else belongs in a *src/modules/perl/foo.c* public API.

1.4.2 xs/maps

The *xs/maps* directory includes mapping files which describe how Apache Perl API should be constructed and various XS typemapping.

These files get modified whenever:

- a new function is added or the API of the existing one is modified.
- a new struct is added or the existing one is modified
- a new C datatype or Perl typemap is added or an existing one is modified.

The execution of:

```
% make source_scan
```

or:

```
% perl build/source_scan.pl
```

converts these map files into their Perl table representation in the *xs/tables/current/* directory. This Perl representation is then used during `perl Makefile.PL` to generate the XS code in the *./WrapXS/* directory by the xs_generate() function. This XS code is combined of the Apache API Perl glue and mod_perl specific extensions.

NOTE: source_scan requires C::Scan 0.75, which at the moment is unreleased, there is a working copy here: http://perl.apache.org/~dougm/Scan.pm

If you need to skip certain unwanted C defines from being picked by the source scanning you can add them to the array `$Apache::ParseSource::defines_unwanted` in *lib/Apache/ParseSource.pm*.

Notice that *source_scan* target is normally not run during the project build process, since the source scanning is not stable yet, therefore everytime the map files change, `make source_scan` should be run manually and the updated files ending up in the *xs/tables/current/* directory should be committed to the cvs repository.

The *source_scan* make target is actually to run *build/source_scan.pl*, which can be run directly without needing to create *Makefile* first.

There are three different types of map files in the *xs/maps/* directory:

- **Functions Mapping**

  ```
  apache_functions.map
  modperl_functions.map
  apr_functions.map
  ```

- **Structures Mapping**

  ```
  apache_structures.map
  apr_structures.map
  ```

- **Types Mapping**

  ```
  apache_types.map
  apr_types.map
  modperl_types.map
  ```

The following sections describe the syntax of the files in each group

1.4.2.1 Functions Mapping

The functions mapping file is comprised of groups of function definitions. Each group starts with a header similar to XS syntax:

```
MODULE=... PACKAGE=... PREFIX=... BOOT=... ISA=...
```

where:

- **`MODULE`**

  the module name where the functions should be put. e.g. `MODULE Apache::Connection` will place the functions into *WrapXS/Apache/Connection.{pm,xs}*.

- **`PACKAGE`**

  the package name functions belong to, defaults to `MODULE`. The value of *guess* indicates that package name should be guessed based on first argument found that maps to a Perl class. If the value is not defined and the function's name starts with *ap_* the `Apache` package will be used, if it starts with *apr_* then the `APR` package is used.

- **`PREFIX`**

  prefix string to be stripped from the function name. If not specified it defaults to `PACKAGE`, converted to C name convention, e.g. `APR::Base64` makes the prefix: *apr_base64_*. If the converted prefix does not match, defaults to *ap_* or *apr_*.

- **BOOT**

  The BOOT directive tells the XS generator, whether to add the boot function to the autogenerated XS file or not. If the value of BOOT is not true or it's simply not declared, the boot function won't be added.

  If the value is true, a boot function will be added to the XS file. Note, that this function is not declared in the map file.

  The boot function name must be constructed from three parts:

  ```
  'mpxs_' . MODULE . '_BOOT'
  ```

  where MODULE is the one declared with MODULE= in the map file.

  For example if we want to have an XS boot function for a class APR::IO, we create this function in *xs/APR/IO/APR__IO.h*:

  ```
  static void mpxs_APR__IO_BOOT(pTHX)
  {
     /* boot code here */
  }
  ```

  and now we add the BOOT=1 declaration to the *xs/maps/modperl_functions.map* file:

  ```
  MODULE=APR::IO PACKAGE=APR::IO BOOT=1
  ```

  Notice that the PACKAGE= declaration is a must.

  When *make xs_generate* is run (after running *make source_scan*), it autogenerates *Wrap/APR/IO/IO.xs* and amongst other things will include:

  ```
  BOOT:
      mpxs_APR__IO_BOOT(aTHXo);
  ```

- **ISA**

  META: complete

Every function definition is declared on a separate line (use \ if the line is too long), using the following format:

```
C function name | Dispatch function name | Argspec | Perl alias
```

where:

- **C function name**

  The name of the real C function.

Function names that do not begin with `/^\w/` are skipped. For details see: `%ModPerl::MapUtil::disabled_map`.

The return type can be specified before the C function name. It defaults to *return_type* in `{Apache,ModPerl}::FunctionTable`.

META: DEFINE nuances

- **Dispatch function name**

  Dispatch function name defaults to C function name. If the dispatch name is just a prefix (*mpxs_*, *MPXS_*) the C function name is appended to it.

  See the explanation about function naming and arguments passing.

- **Argspec**

  The argspec defaults to arguments in `{Apache,ModPerl}::FunctionTable`. Argument types can be specified to override those in the `FunctionTable`. Default values can be specified, e.g. `arg=default_value`. Argspec of `...` indicates *passthru*, calling the function with `(aTHX_ I32 items, SP **sp, SV **MARK)`.

- **Perl alias**

  the Perl alias will be created in the current `PACKAGE`.

1.4.2.2 Structures Mapping

META: complete

1.4.2.3 Types Mapping

META: complete

1.4.2.4 Modifying Maps

As explained in the beginning of this section, whenever the map file is modified you need first to run:

```
% make source_scan
```

Next check that the conversion to Perl tables is properly done by verifying the resulting corresponding file in *xs/tables/current*. For example *xs/maps/modperl_functions.map* is converted into *xs/tables/current/ModPerl/FunctionTable.pm*.

If you want to do a visual check on how XS code will be generated, run:

```
% make xs_generate
```

and verify that the autogenerated XS code under the directory *./WrapXS* is correct. Notice that for functions, whose arguments or return types can't be resolved, the XS glue won't be generated and a warning will be printed. If that's the case add the missing type's typemap to the types map file as explained in Adding Typemaps for new C Data Types and run the XS generation stage again.

You can also build the project normally:

```
% perl Makefile.PL ...
```

which runs the XS generation stage.

1.4.3 XS generation process

As mentioned before XS code is generated in the *WrapXS* directory either during `perl Makefile.PL` via xs_generate() if `MP_GENERATE_XS=1` is used (which is the default) or explicitly via:

```
% make xs_generate
```

In addition it creates a number of files in the *xs/* directory:

```
modperl_xs_sv_convert.h
modperl_xs_typedefs.h
```

1.5 Gluing Existing APIs

If you have an API that you simply want to provide the Perl interface without writing any code...

META: complete

WrapXS allows you to adjust some arguments and supply default values for function arguments without writing any code

META: complete

`MPXS_` functions are final `XSUBs` and always accept:

```
aTHX_ I32 items, SP **sp, SV **MARK
```

as their arguments. Whereas `mpxs_` functions are either intermediate thin wrappers for the existing C functions or functions that do something by themselves. `MPXS_` functions also can be used for writing thin wrappers for C macros.

1.6 Adding Wrappers for existing APIs and Creating New APIs

In certain cases the existing APIs need to be adjusted. There are a few reasons for doing this.

First, is to make the given C API more Perlish. For example C functions cannot return more than one value, and the pass by reference technique is used. This is not Perlish. Perl has no problem returning a list of value, and passing by reference is used only when an array or a hash in addition to any other variables need to be passes or returned from the function. Therefore we may want to adjust the C API to return a list rather than passing a reference to a return value, which is not intuitive for Perl programmers.

Second, is to adjust the functionality, i.e. we still use the C API but may want to adjust its arguments before calling the original function, or do something with return values. And of course optionally adding some new code.

Third, is to create completely new APIs. It's quite possible that we need more functionality built on top of the existing API. In that case we simply create new APIs.

The following sections discuss various techniques for retrieving function arguments and returning values to the caller. They range from using usual C argument passing and returning to more complex Perl arguments' stack manipulation. Once you know how to retrieve the arguments in various situations and how to put the return values on the stack, the rest is usually normal C programming potentially involving using Perl APIs.

Let's look at various ways we can declare functions and what options various declarions provide to us:

1.6.1 Functions Returning a Single Value (or Nothing)

If its know deterministically what the function returns and there is only a single return value (or nothing is returned == *void*), we are on the C playground and we don't need to manipulate the returning stack. However if the function may return a single value or nothing at all, depending on the inputs and the code, we have to manually manipulate the stack and therefore this section doesn't apply.

Let's look at various requirements and implement these using simple examples. The following testing code exercises the interfaces we are about to develop, so refer to this code to see how the functions are invoked from Perl and what is returned:

```
file:t/response/TestApache/coredemo.pm
---------------------------------------
package TestApache::coredemo;

use strict;
use warnings FATAL => 'all';

use Apache::Const -compile => 'OK';

use Apache::Test;
use Apache::TestUtil;

use Apache::CoreDemo;

sub handler {
    my $r = shift;
```

```
    plan $r, tests => 7;

    my $a = 7;
    my $b = 3;
    my ($add, $subst);

    $add = Apache::CoreDemo::print($a, $b);
    t_debug "print";
    ok !$add;

    $add = Apache::CoreDemo::add($a, $b);
    ok t_cmp($a + $b, $add, "add");

    $add = Apache::CoreDemo::add_sv($a, $b);
    ok t_cmp($a + $b, $add, "add: return sv");

    $add = Apache::CoreDemo::add_sv_sv($a, $b);
    ok t_cmp($a + $b, $add, "add: pass/return svs");

    ($add, $subst) = @{ Apache::CoreDemo::add_subst($a, $b) };
    ok t_cmp($a + $b, $add,   "add_subst: add");
    ok t_cmp($a - $b, $subst, "add_subst: subst");

    $subst = Apache::CoreDemo::subst_sp($a, $b);
    ok t_cmp($a - $b, $subst, "subst via SP");

    Apache::OK;
}

1;
```

The first case is the simplest: pass two integer arguments, print these to the STDERR stream and return nothing:

```
file:xs/Apache/CoreDemo/Apache__CoreDemo.h
-------------------------------------------
static MP_INLINE
void mpxs_Apache__CoreDemo_print(int a, int b)
{
    fprintf(stderr, "%d, %d\n", a, b);
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache::CoreDemo
 mpxs_Apache__CoreDemo_print
```

Now let’s say that the *b* argument is optional and in case it wasn’t provided, we want to use a default value, e.g. 0. In that case we don’t need to change the code, but simply adjust the map file to be:

```
file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache::CoreDemo
 mpxs_Apache__CoreDemo_print | | a, b=0
```

In the previous example, we didn't list the arguments in the map file since they were automatically retrieved from the source code. In this example we tell WrapXS to assign a value of `0` to the argument b, if it wasn't supplied by the caller. All the arguments must be listed and in the same order as they are defined in the function.

You may add an extra test that test teh default value assignment:

```
$add = Apache::CoreDemo::add($a);
ok t_cmp($a + 0, $add, "add (b=0 default)");
```

The second case: pass two integer arguments and return their sum:

```
file:xs/Apache/CoreDemo/Apache__CoreDemo.h
------------------------------------------
static MP_INLINE
int mpxs_Apache__CoreDemo_add(int a, int b)
{
    return a + b;
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache::CoreDemo
 mpxs_Apache__CoreDemo_add
```

The third case is similar to the previous one, but we return the sum as as a Perl scalar. Though in C we say SV*, in the Perl space we will get a normal scalar:

```
file:xs/Apache/CoreDemo/Apache__CoreDemo.h
------------------------------------------
static MP_INLINE
SV *mpxs_Apache__CoreDemo_add_sv(pTHX_ int a, int b)
{
    return newSViv(a + b);
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache::CoreDemo
 mpxs_Apache__CoreDemo_add_sv
```

In the second example the XSUB function was converting the returned *int* value to a Perl scalar behind the scenes. In this example we return the scalar ourselves. This is of course to demonstrate that you can return a Perl scalar, which can be a reference to a complex Perl datastructure, which we will see in the fifth example.

The forth case demonstrates that you can pass Perl variables to your functions without needing XSUB to do the conversion. In all previous examples XSUB was automatically converting Perl scalars in the argument list to the corresponding C variables, using the typemap definitions.

```
file:xs/Apache/CoreDemo/Apache__CoreDemo.h
-------------------------------------------
static MP_INLINE
SV *mpxs_Apache__CoreDemo_add_sv_sv(pTHX_ SV *a_sv, SV *b_sv)
{
    int a = (int)SvIV(a_sv);
    int b = (int)SvIV(b_sv);

    return newSViv(a + b);
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache::CoreDemo
 mpxs_Apache__CoreDemo_add_sv_sv
```

So this example is the same simple case of addition, though we manually convert the Perl variables to C variables, perform the addition operation, convert the result to a Perl Scalar of kind *IV* (Integer Value) and return it directly to the caller.

In case where more than one value needs to be returned, we can still implement this without directly manipulating the stack before a function returns. The fifth case demonstrates a function that returns the result of addition and substruction operations on its arguments:

```
file:xs/Apache/CoreDemo/Apache__CoreDemo.h
-------------------------------------------
static MP_INLINE
SV *mpxs_Apache__CoreDemo_add_subst(pTHX_ int a, int b)
{
    AV *av = newAV();

    av_push(av, newSViv(a + b));
    av_push(av, newSViv(a - b));

    return newRV_noinc((SV*)av);
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache::CoreDemo
 mpxs_Apache__CoreDemo_add_subst
```

If you look at the corresponding testing code:

```
    ($add, $subst) = @{ Apache::CoreDemo::add_subst($a, $b) };
    ok t_cmp($a + $b, $add,   "add_subst: add");
    ok t_cmp($a - $b, $subst, "add_subst: subst");
```

you can see that this technique comes at a price of needing to dereference the return value to turn it into a list. The actual code is very similar to the `Apache::CoreDemo::add_sv` function which was doing only the addition operation and returning a Perl scalar. Here we perform the addition and the substraction operation and push the two results into a previously created *AV** data structure, which represents an array. Since only the *SV* datastructures are allowed to be put on stack, we take a reference *RV* (which is of an *SV* kind) to the existing *AV* and return it.

The sixth case demonstrates a situation where the number of arguments or their types may vary and aren't known at compile time. Though notice that we still know that we are returning at compile time (zero or one arguments), *int* in this example:

```
file:xs/Apache/CoreDemo/Apache__CoreDemo.h
------------------------------------------
static MP_INLINE
int mpxs_Apache__CoreDemo_subst_sp(pTHX_ I32 items, SV **MARK, SV **SP)
{
    int a, b;

    if (items != 2) {
        Perl_croak(aTHX_ "usage: ...");
    }

    a = mp_xs_sv2_int(*MARK);
    b = mp_xs_sv2_int(*(MARK+1));

    return a - b;
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache::CoreDemo
 mpxs_Apache__CoreDemo_subst_sp | | ...
```

In the map file we use a special token `...` which tells the XSUB constructor to pass `items`, `MARK` and `SP` arguments to the function. The macro `MARK` points to the first argument passed by the caller in the Perl namespace. For example to access the second argument to retrieve the value of `b` we use `*(MARK+1)`, which if you remember represented as an *SV* variable, which nees to be converted to the corresponding C type.

In this example we use the macro *mp_xs_sv2_int*, automatically generated based on the data from the *xs/typemap* and *xs/maps/*_types.map* files, and placed into the *xs/modperl_xs_sv_convert.h* file. In the case of *int* C type the macro is:

```
#define mp_xs_sv2_int(sv) (int)SvIV(sv)
```

which simply converts the *SV* variable on the stack and generates an *int* value.

While in this example you have an access to the stack, you cannot manipulate the return values, because the XSUB wrapper expects a single return value of type *int*, so even if you put something on the stack it will be ignored.

1.6.2 Functions Returning Variable Number of Values

We saw earlier that if we want to return an array one of the ways to go is to return a reference to an array as a single return value, which fits the C paradigm. So we simply declare the return value as `SV*`.

This section talks about cases where it's unknown at compile time how many return values will be or it's known that there will be more than one return value--something that C cannot handle via its return mechanism.

Let's rewrite the function `mpxs_Apache__CoreDemo_add_subst` from the earlier section to return two results instead of a reference to a list:

```
file:xs/Apache/CoreDemo/Apache__CoreDemo.h
-------------------------------------------
static XS(MPXS_Apache__CoreDemo_add_subst_sp)
{
    dXSARGS;
    int a, b;

    if (items != 2) {
        Perl_croak(aTHX_ "usage: Apache::CoreDemo::add_subst_sp($a, $b)");
    }
    a = mp_xs_sv2_int(ST(0));
    b = mp_xs_sv2_int(ST(1));

    SP -= items;

    if (GIMME == G_ARRAY) {
      EXTEND(sp, 2);
      PUSHs(sv_2mortal(newSViv(a + b)));
      PUSHs(sv_2mortal(newSViv(a - b)));
    }
    else {
        XPUSHs(sv_2mortal(newSViv(a + b)));
    }

    PUTBACK;
}
```

Before explaining the function here is the prototype we add to the map file:

```
file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache::CoreDemo
DEFINE_add_subst_sp | MPXS_Apache__CoreDemo_add_subst_sp | ...
```

The `mpxs_` functions declare in the third column the arguments that they expect to receive (and optionally the default values). The `MPXS` functions are the real `XSUBs` and therefore they always accept:

```
aTHX_ I32 items, SP **sp, SV **MARK
```

as their arguments. Thefore it doesn't matter what is placed in this column when the `MPXS_` function is declared. Usually for documentation the Perl side arguments are listed. For example you can say:

```
DEFINE_add_subst_sp | MPXS_Apache__CoreDemo_add_subst_sp | x, y
```

In this function we manually manipulate the stack to retrieve the arguments passed on the Perl side and put the results back onto the stack. Therefore the first thing we do is to initialize a few special variables using the `dXSARGS` macro defined in *XSUB.h*, which in fact calls a bunch of other macros. These variables help

to manipulate the stack. dSP is one of these macros and it declares and initial- izes a local copy of the Perl stack pointer sp which . This local copy should always be accessed as SP.

We retrieve the original function arguments using the ST() macros. ST(0) and ST(1) point to the first and the second argument on the stack, respectively. But first we check that we have exactly two arguments on the stack, and if not we abort the function. The items variable is the function argument.

Once we have retrieved all the arguments from the stack we set the local stack pointer SP to point to the bottom of the stack (like there are no items on the stack):

```
SP -= items;
```

Now we can do whatever processing is needed and put the results back on the stack. In our example we return the results of addition and substraction operations if the function is called in the list context. In the scalar context the function returns only the result of the addition operation. We use the GIMME macro which tells us the context.

In the list context we make sure that we have two spare slots on the stack since we are going to push two items, and then we push them using the PUSHs macro:

```
EXTEND(sp, 2);
PUSHs(sv_2mortal(newSViv(a + b)));
PUSHs(sv_2mortal(newSViv(a - b)));
```

Alternatively we could use:

```
XPUSHs(sv_2mortal(newSViv(a + b)));
XPUSHs(sv_2mortal(newSViv(a - b)));
```

The XPUSHs macro e*X*tends the stack before pushing the item into it if needed. If we plan to push more than a single item onto the stack, it's more efficient to extend the stack in one call.

In the scalar context we push only one item, so here we use the XPUSHs macro:

```
XPUSHs(sv_2mortal(newSViv(a + b)));
```

The last command we call is:

```
PUTBACK;
```

which makes the local stack pointer global. This is a must call if the state of the stack was changed when the function is about to return. The stack changes if something was popped from or pushed to it, or both and changed the number of items on the stack.

In our example we don't need to call PUTBACK if the function is called in the list context. Because in this case we return two variables, the same as two function arguments, the count didn't change. Though in the scalar context we push onto the stack only one argument, so the function won't return what is expected. The simplest way to avoid errors here is to always call PUTBACK when the stack is changed.

For more information refer to the *perlcall* manpage which explains the stack manipulation process in great details.

Finally we test the function in the list and scalar contexts:

```
file:t/response/TestApache/coredemo.pm
---------------------------------------
...
my $a = 7;
my $b = 3;
my ($add, $subst);

# list context
($add, $subst) = Apache::CoreDemo::add_subst_sp($a, $b);
ok t_cmp($a + $b, $add,   "add_subst_sp list context: add");
ok t_cmp($a - $b, $subst, "add_subst_sp list context: subst");

# scalar context
$add = Apache::CoreDemo::add_subst_sp($a, $b);
ok t_cmp($a + $b, $add,   "add_subs_spt scalar context: add");
...
```

1.6.3 Wrappers Functions for C Macros

Let's say you have a C macro which you want to provide a Perl interface for. For example let's take a simple macro which performs the power of function:

```
file:xs/Apache/CoreDemo/Apache__CoreDemo.h
------------------------------------------
#define mpxs_Apache__CoreDemo_power(x, y) pow(x, y)
```

To create the XS glue code we use the following entry in the map file:

```
file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache::CoreDemo
double:DEFINE_power | | double:x, double:y
```

This works very similar to the `MPXS_Apache__CoreDemo_add_subst_sp` function presented earlier. But since this is a macro the XS wrapper needs to know the types of the arguments and the return type, so these are added. The return type is added just before the function name and separated from it by the colon (`:`), the argument types are specified in the third column. The type is always separated from the name of the variable by the colon (`:`).

And of course finally we need to test that the function works in Perl:

```
file:t/response/TestApache/coredemo.pm
--------------------------------------
...
my $a = 7;
my $b = 3;
my $power = Apache::CoreDemo::power($a, $b);
ok t_cmp($a ** $b, $power, "power macro");
...
```

1.7 Wrappers for modperl_, apr_ and ap_ APIs

If you already have a C function whose name starts from *modperl_*, *apr_* or *ap_* and you want to do something before calling the real C function, you can write a XS wrapper using the same method as in the MPXS_Apache__CoreDemo_add_subst_sp . The only difference is that it'll be clearly seen in the map file that this is a wrapper for an existing C API.

Let's say that we have an existing C function apr_power(), this is how we declare its wrapper:

```
file:xs/maps/apr_functions.map
----------------------------------
MODULE=APR::Foo
apr_power | MPXS_ | x, y
```

The first column specifies the existing function's name, the second tells that the XS wrapper will use the `MPXS_` prefix, which means that the wrapper must be called `MPXS_apr_power`. The third column specifies the argument names, but for `MPXS_` no matter what you specify there the `...` will be passed:

```
aTHX_ I32 items, SP **sp, SV **MARK
```

so you can leave that column empty, but here we use `x` and `y` to remind us that these two arguments are passed from Perl.

If the forth column is empty this function will be called `APR::Foo::power` in the Perl namespace. But you can use that column to give a different Perl name, e.g with:

```
apr_power | MPXS_ | x, y | pow
```

This function will be available from Perl as `APR::Foo::pow`.

Similarly you can write a `MPXS_modperl_power` wrapper for a `modperl_power()` function but here you have to explicitly give the Perl function's name in the forth column:

```
file:xs/maps/apr_functions.map
----------------------------------
MODULE=Apache::CoreDemo
modperl_power | MPXS_ | x, y | mypower
```

and the Perl function will be called `Apache::CoreDemo::mypower`.

The `MPXS_` wrapper's implementation is similar to MPXS_Apache__CoreDemo_add_subst_sp .

1.8 MP_INLINE vs C Macros vs Normal Functions

To make the code maintainable and reusable functions and macros are used in when programming in C (and other languages :).

When function is marked as *inlined* it's merely a hint to the compiler to replace the call to a function with the code inside this function (i.e. inlined). Not every function can be inlined. Some typical reasons why inlining is sometimes not done include:

- the function calls itself, that is, is recursive
- the function contains loops such as `for(;;)` or `while()`
- the function size is too large

Most of the advantage of inline functions comes from avoiding the overhead of calling an actual function. Such overhead includes saving registers, setting up stack frames, etc. But with large functions the overhead becomes less important.

Use the `MP_INLINE` keyword in the declaration of the functions that are to be inlined. The functions should be inlined when:

- Only ever called once (the *wrappers* that are called from *.xs* files), no matter what the size of code is.
- Short bodies of code called in a *hot* code (like *modperl_env_hv_store*, which is called many times inside of a loop), where it is cleaner to see the code in function form rather than macro with lots of \'s. Remember that an inline function takes much more space than a normal functions if called from many places in the code.

Of course C macros are a bit faster then inlined functions, since there is not even *short jump* to be made, the code is literally copied into the place it's called from. However using macros comes at a price:

- Also unlike macros, in functions argument types are checked, and necessary conversions are performed correctly. With macros it's possible that weird things will happen if the caller has passed arguments of the wrong type when calling a macro.
- One should be careful to pass only absolute values as *"arguments"* to macros. Consider a macro that returns an absolute value of the passed argument:

  ```
  #define ABS(v) ( (v) >= 0 ? (v) : -(v) )
  ```

  In our example if you happen to pass a function it will be called twice:

  ```
  abs_val = ABS(f());
  ```

Since it'll be extended as:

```
abs_val = f() >= 0 ? f() : -f();
```

You cannot do simple operation like increment--in our example it will be called twice:

```
abs_val = ABS(i++);
```

Because it becomes:

```
abs_val = i++ >= 0 ? i++ : -i++;
```

- It's dangerous to use the if() condition without enclosing the code in `{}`, since the macro may be called from inside another if-else condition, which may cause the else part called if the if() part from the macro fails.

  But we always use `{}` for the code inside the if-else condition, so it's not a problem here.

- A multi-line macro can cause problems if someone uses the macro in a context that demands a single statement.

  ```
  while (foo) MYMACRO(bar);
  ```

  But again, we always enclose any code in conditional with `{}`, so it's not a problem for us.

- Inline functions present a problem for debuggers and profilers, because the function is expanded at the point of call and loses its identity. This makes the debugging process a nightmare.

  A compiler will typically have some option available to disable inlining.

In all other cases use normal functions.

1.9 Adding New Interfaces

1.9.1 Adding Typemaps for new C Data Types

Sometimes when a new interface is added it may include C data types for which we don't have corresponding XS typemaps yet. In such a case, the first thing to do is to provide the required typemaps.

Let's add a prototype for the *typedef struct scoreboard* data type defined in *httpd-2.0/include/scoreboard.h*.

First we include the relevant header files in *src/modules/perl/modperl_apache_includes.h*:

```
#include "scoreboard.h"
```

If you want to specify your own type and don't have a header file for it (e.g. if you extend some existing datatype within mod_perl) you may add the *typedef* to *src/modules/perl/modperl_types.h*.

After deciding that `Apache::Scoreboard` is the Perl class will be used for manipulating C *scoreboard* data structures, we map the *scoreboard* data structure to the `Apache::Scoreboard` class. Therefore we add to *xs/maps/apache_types.map*:

```
struct scoreboard       | Apache::Scoreboard
```

Since we want the *scoreboard* data structure to be an opaque object on the perl side, we simply let mod_perl use the default `T_PTROBJ` typemap. After running `make xs_generate` you can check the assigned typemap in the autogenerated *WrapXS/typemap* file.

If you need to do some special handling while converting from C to Perl and back, you need to add the conversion functions to the *xs/typemap* file. For example the `Apache::RequestRec` objects need special handling, so you can see the special `INPUT` and `OUTPUT` typemappings for the corresponding `T_APACHEOBJ` object type.

Now we run `make xs_generate` and find the following definitions in the autogenerated files:

```
file:xs/modperl_xs_typedefs.h
-----------------------------
typedef scoreboard * Apache__Scoreboard;

file:xs/modperl_xs_sv_convert.h
-------------------------------
#define mp_xs_sv2_Apache__Scoreboard(sv) \
((SvROK(sv) && (SvTYPE(SvRV(sv)) == SVt_PVMG)) \
|| (Perl_croak(aTHX_ "argument is not a blessed reference \
(expecting an Apache::Scoreboard derived object)"),0) ? \
(scoreboard *)SvIV((SV*)SvRV(sv)) : (scoreboard *)NULL)

#define mp_xs_Apache__Scoreboard_2obj(ptr) \
sv_setref_pv(sv_newmortal(), "Apache::Scoreboard", (void*)ptr)
```

The file *xs/modperl_xs_typedefs.h* declares the typemapping from C to Perl and equivalent to the `TYPEMAP` section of the XS's *typemap* file. The second file *xs/modperl_xs_sv_convert.h* generates two macros. The first macro is used to convert from Perl to C datatype and equivalent to the *typemap* file's `INPUT` section. The second macro is used to convert from C to Perl datatype and equivalent to the *typemap*'s `OUTPUT` section.

Now proceed on adding the glue code for the new interface.

1.9.2 Importing Constants and Enums into Perl API

To *import* httpd and APR constants and enums into Perl API, edit *lib/Apache/ParseSource.pm*. To add a new type of `DEFINE` constants adjust the `%defines_wanted` variable, for `enums` modify `%enums_wanted`.

For example to import all `DEFINE`s starting with `APR_FLOCK_` add:

```
my %defines_wanted = (
    ...
    APR => {
        ...
        flock     => [qw{APR_FLOCK_}],
        ...
    },
);
```

When deciding which constants are to be exported, the regular expression will be used, so in our example all matches `/^APR_FLOCK_/` will be imported into the Perl API.

For example to import an *read_type_e* `enum` for APR, add:

```
my %enums_wanted = (
    APR => { map { $_, 1 } qw(apr_read_type) },
);
```

Notice that *_e* part at the end of the enum name has gone.

After adding/modifying the datastructures make sure to run `make source_scan` or `perl build/source_scan.pl` and verify that the wanted constant or enum were picked by the source scanning process. Simply grep *xs/tables/current* for the wanted string. For example after adding *apr_read_type_e* enum we can check:

```
% more xs/tables/current/Apache/ConstantsTable.pm
...
  'read_type' => [
    'APR_BLOCK_READ',
    'APR_NONBLOCK_READ'
  ],
```

Of course the newly added constant or enum's typemap should be declared in the appropriate *xs/maps/*_types.map* files, so the XS conversion of arguments will be performed correctly. For example *apr_read_type* is an APR enum so it's declared in *xs/maps/apr_types.map*:

```
apr_read_type          | IV
```

`IV` is used as a typemap, Since enum is just an integer. In more complex cases the typemap can be different. (META: examples)

1.10 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

Stas Bekman <stas (at) stason.org>

1.11 Authors

- Stas Bekman <stas (at) stason.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents:

1 mod_perl 2.0 Source Code Explained 1
1.1 Description 2
1.2 Project's Filesystem Layout 2
1.3 Directory src 2
1.3.1 Directory src/modules/perl/ 2
1.4 Directory xs/ 2
1.4.1 xs/Apache, xs/APR and xs/ModPerl 3
1.4.2 xs/maps 3
1.4.2.1 Functions Mapping 4
1.4.2.2 Structures Mapping 6
1.4.2.3 Types Mapping 6
1.4.2.4 Modifying Maps 6
1.4.3 XS generation process 7
1.5 Gluing Existing APIs 7
1.6 Adding Wrappers for existing APIs and Creating New APIs 7
1.6.1 Functions Returning a Single Value (or Nothing) 8
1.6.2 Functions Returning Variable Number of Values 12
1.6.3 Wrappers Functions for C Macros 15
1.7 Wrappers for modperl_, apr_ and ap_ APIs 16
1.8 MP_INLINE vs C Macros vs Normal Functions 17
1.9 Adding New Interfaces 18
1.9.1 Adding Typemaps for new C Data Types 18
1.9.2 Importing Constants and Enums into Perl API 19
1.10 Maintainers 20
1.11 Authors 21