Extension Writing Part I: Introduction to Php and Zend

Introduction

If you're reading this tutorial, you probably have some interest in writing an extension for the PHP language. If not… well perhaps when we're done you'll have discovered an interest you didn't know existed!

This tutorial assumes basic familiarity with both the PHP language and the language the PHP interpreter is written in: C.

Let's start by identifying why you might want to write a PHP extension.

There is some library or OS-specific call which cannot be made from PHP directly because of the degree of abstraction inherent in the language.

You want to make PHP itself behave in some unusual way.

You've already got some PHP code written, but you know it could be faster, smaller, and consume less memory while running.

You have a particularly clever bit of code you want to sell, and it's important that the party you sell it to be able to execute it, but not view the source.

These are all perfectly valid reasons, but in order to create an extension, you need to understand what an extension is first.

What's an Extension?

If you've used PHP, you've used extensions. With only a few exceptions, every userspace function in the PHP language is grouped into one extension or another. A great many of these functions are part of the standard extension – over 400 of them in total. The PHP source bundle comes with around 86 extensions, having an average of about 30 functions each. Do the math, that's about 2500 functions. As if this weren't enough, the PECL repository offers over 100 additional extensions, and even more can be found elsewhere on the Internet.

"With all these functions living in extensions, what's left?" I hear you ask.

“What are they an extension to? What is the ‘core’ of PHP?”

PHP’s core is made up of two separate pieces. At the lowest levels you find the Zend Engine (ZE). ZE handles parsing a human-readable script into machine-readable tokens, and then executing those tokens within a process space. ZE also handles memory management, variable scope, and dispatching function calls. The other half of this split personality is the PHP core. PHP handles communication with, and bindings to, the SAPI layer (Server Application Programming Interface, also commonly used to refer to the host environment – Apache, IIS, CLI, CGI, etc). It also provides a unified control layer for safe_mode and open_basedir checks, as well as the streams layer which associates file and network I/O with userspace functions like fopen(), fread(), and fwrite().

Lifecycles

When a given SAPI starts up, for example in response to /usr/local/apache/bin/apachectl start, PHP begins by initializing its core subsystems. Towards the end of this startup routine, it loads the code for each extension and calls their Module Initialization routine (MINIT). This gives each extension a chance to initialize internal variables, allocate resources, register resource handlers, and register its functions with ZE, so that if a script calls one of those functions, ZE knows which code to execute.

Next, PHP waits for the SAPI layer to request a page to be processed. In the case of the CGI or CLI SAPIs, this happens immediately and only once. In the case of Apache, IIS, or other fully-fledged web server SAPIs, it occurs as pages are requested by remote users and repeats any number of times, possibly concurrently. No matter how the request comes in, PHP begins by asking ZE to setup an environment for the script to run in, then calls each extension’s Request Initialization (RINIT) function. RINIT gives the extension a chance to set up specific environment variables, allocate request specific resources, or perform other tasks such as auditing. A prime example of the RINIT function in action is in the sessions extension where, if the session.auto_start option is enabled, RINIT will automatically trigger the userspace session_start() function and pre-populate the $_SESSION variable.

Once the request is initialized, ZE takes over by translating the PHP script into tokens, and finally to opcodes which it can step through and execute. Should one of these opcodes require an extension function to be called, ZE will bundle up the arguments for that function, and temporarily give over control until it completes.

After a script has finished executing, PHP calls the Request Shutdown (RSHUTDOWN) function of each extension to perform any last minute cleanup (such as saving session variables to disk). Next, ZE performs a cleanup process (known as garbage collection) which effectively performs

an unset() on every variable used during the previous request.

Once completed, PHP waits for the SAPI to either request another document or signal a shutdown. In the case of the CGI and CLI SAPIs, there is no “next request”, so the SAPI initiates a shutdown immediately. During shutdown, PHP again cycles through each extension calling their Module Shutdown (MSHUTDOWN) functions, and finally shuts down its own core subsystems.

This process may sound daunting at first, but once you dive into a working extension it should all gradually start to make sense.

Memory Allocation

In order to avoid losing memory to poorly written extensions, ZE performs its own internal memory management using an additional flag that indicates persistence. A persistent allocation is a memory allocation that is meant to last for longer than a single page request. A non-persistent allocation, by contrast, is freed at the end of the request in which it was allocated, whether or not the free function is called. Userspace variables, for example, are allocated non-persistently because at the end of a request they’re no longer useful.

While an extension may, in theory, rely on ZE to free non-persistent memory automatically at the end of each page request, this is not recommended. Memory allocations will remain unreclaimed for longer periods of time, resources associated with that memory will be less likely to be shutdown properly, and it’s just poor practice to make a mess without cleaning it up. As you’ll discover later on, it’s actually quite easy to ensure that all allocated data is cleaned up properly

Let’s briefly compare traditional memory allocation functions (which should only be used when working with external libraries) with persistent and non-persistent memory allocation within PHP/ZE.

Traditional Non-Persistent Persistent

malloc(count)

calloc(count, num) emalloc(count)

ecalloc(count, num) pemalloc(count, 1)*

pecalloc(count, num, 1)

strdup(str)

strndup(str, len) estrdup(str)

estrndup(str, len) pestrdup(str, 1)

pemalloc() & memcpy()

free(ptr) efree(ptr) pefree(ptr, 1)

realloc(ptr, newsize) erealloc(ptr, newsize) perealloc(ptr, newsize, 1)

malloc(count * num + extr)** safe_emalloc(count, num, extr) safe_pemalloc(count, num, extr)

* The pemalloc() family include a ‘persistent’ flag which allows

them to behave like their non-persistent counterparts.

For example:

emalloc(1234) is the same as pemalloc(1234, 0) ** safe_emalloc() and (in PHP 5) safe_pemalloc() perform an additional check to avoid integer overflows

Setting Up a Build Environment

Now that you’ve covered some of the theory behind the workings of PHP and the Zend Engine, I’ll bet you’d like to dive in and start building something. Before you can do that however, you’ll need to collect some necessary build tools and set up an environment suited to your purposes.

First, you’ll need PHP itself, and the set of build tools required by PHP. If you’re unfamiliar with building PHP from source, I suggest you take a look at http://www.php.net/install.unix. (Developing PHP extensions for Windows will be covered in a later article). While it might be tempting to use a binary package of PHP from your distribution of choice, these versions tend to leave out two important ./configure options that are very handy during the development process. The first is --enable-debug. This option will compile PHP with additional symbol information loaded into the executable so that, if a segfault occurs, you’ll be able to collect a core dump from it and use gdb to track down where the segfault occurred and why. The other option depends on which version of PHP you’ll be developing against. In PHP 4.3 this option is named --enable-experimental-zts, in PHP 5 and later it’s --enable-maintainer-zts. This option will make PHP think its operating in a multi-threaded

environment and will allow you to catch common programming mistakes which, while harmless in a non-threaded environment, will cause your extension to be unusable in a multi-threaded one. Once you've compiled PHP using these extra options and installed it on your development server (or workstation), you can begin to put together your first extension.

Hello World

What programming introduction would be complete without the requisite Hello World application? In this case, you'll be making an extension that exports a single function returning a string containing the words: "Hello World". In PHP code you'd probably do it something like this

Now you're going to turn that into a PHP extension. First let's create a directory called hello under the ext/ directory in your PHP source tree and chdir into that folder. This directory can actually live anywhere inside or outside the PHP tree, but I'd like you to place it here to demonstrate an unrelated concept in a later article. Here you need to create three files: a source file containing your hello_world function, a header file containing references used by PHP to load your extension, and a configuration file used by phpize to prepare your extension for compiling.

config.m4

PHP_ARG_ENABLE(hello, whether to enable Hello World support, [--enable-hello Enable Hello World support]) if test "$PHP_HELLO" = "yes"; then AC_DEFINE(HAVE_HELLO, 1, [Whether you have Hello World]) PHP_NEW_EXTENSION(hello, hello.c, $ext_shared) fi php_hello.h

#ifndef PHP_HELLO_H

#define PHP_HELLO_H 1

#define PHP_HELLO_WORLD_VERSION "1.0"

#define PHP_HELLO_WORLD_EXTNAME "hello"

PHP_FUNCTION(hello_world);

extern zend_module_entry hello_module_entry;

#define phpext_hello_ptr &hello_module_entry

#endif

hello.c

#ifdef HAVE_CONFIG_H

#include "config.h"

#endif

#include "php.h"

#include "php_hello.h

static function_entry hello_functions[] = { PHP_FE(hello_world, NULL) {NULL, NULL, NULL}};

zend_module_entry hello_module_entry = #if ZEND_MODULE_API_NO >= 20010901

STANDARD_MODULE_HEADER,

#endif

PHP_HELLO_WORLD_EXTNAME,

hello_functions,

NULL,

NULL,

NULL,

NULL,

NULL,

#if ZEND_MODULE_API_NO >= 20010901

PHP_HELLO_WORLD_VERSION,

```
#endif

STANDARD_MODULE_PROPERTIES

#ifdef COMPILE_DL_HELLO

ZEND_GET_MODULE(hello)

#endif

PHP_FUNCTION(hello_world)

    RETURN_STRING("Hello World", 1);
```

Most of the code you can see in the example extension above is just glue - protocol language to introduce the extension to PHP and establish a dialogue for them to communicate. Only the last four lines are what you might call “real code” which performs a task on a level that the userspace script might interact with. Indeed the code at this level looks very similar to the PHP code we looked at earlier and can be easily parsed on sight:

Declare a function named hello_world Have that function return a string: “Hello World” ….um…. 1? What’s that 1 all about? Recall that ZE includes a sophisticated memory management layer which ensures? that allocated resources are freed when the script exits. In the land of memory management however, it’s a big no-no to free the same block of memory twice. This action, called double freeing, is a common cause of segmentation faults, as it involves the calling program trying to access a block of memory which it no longer owns. Similarly, you don’t want to allow ZE to free a static string buffer (such as “Hello World” in our example extension) as it lives in program space and thus isn’t a data block to be owned by any one process. RETURN_STRING() could assume that any strings passed to it need to be copied so that they can be safely freed later; but since it’s not uncommon for an internal function to allocate memory for a string, fill it dynamically, then return it, RETURN_STRING() allows us to specify whether it’s necessary to make a copy of the string value or not. To further illustrate this concept, the following code snippet is identical to its counterpart above:

```
PHP_FUNCTION(hello_world)

{ char *str;

 str = estrdup("Hello World");
```

RETURN_STRING(str, 0);}

In this version, you manually allocated the memory for the “Hello World” string that will ultimately be passed back to the calling script, then “gave” that memory to RETURN_STRING(), using a value of 0 in the second parameter to indicate that it didn’t need to make its own copy, it could have ours.

Building Your Extension

The final step in this exercise will be building your extension as a dynamically loadable module. If you’ve copied the example above correctly, this should only take three commands run from ext/hello/:

$ phpize

$ ./configure --enable-hello

$ make

After running each of these commands, you should have a hello.so file in ext/hello/modules/ . Now, as with any other PHP extension, you can just copy this to your extensions directory (/usr/local/lib/php/extensions/ is the default, check your php.ini to be sure) and add the line extension=hello.so to your php.ini to trigger it to load on startup. For CGI/CLI SAPIs, this simply means the next time PHP is run; for web server SAPIs like Apache, this will be the next time the web server is restarted. Let’s give it a try from the command line for now:

$ php -r 'echo hello_world();'

If everything’s gone as it should, you should see Hello World output by this script, since the hello_world() function in your loaded extension returns that string, and the echo command displays whatever is passed to it (the result of the function in this case).

Other scalars may be returned in a similar fashion, using RETURN_LONG() for integer values, RETURN_DOUBLE() for floating point values, RETURN_BOOL() for true/false values, and RETURN_NULL() for, you guessed it, NULL values. Let’s take a look at each of those in action by adding PHP_FE() lines to the function_entry struct in hello.c and adding some PHP_FUNCTION()s to the end of the file.static function_entry hello_functions[] =

{

```
    PHP_FE(hello_world, NULL)
    PHP_FE(hello_long, NULL)
    PHP_FE(hello_double, NULL)
    PHP_FE(hello_bool, NULL)
    PHP_FE(hello_null, NULL)
    {NULL, NULL, NULL}};
PHP_FUNCTION(hello_long)
{ RETURN_LONG(42)}
PHP_FUNCTION(hello_double)
{   RETURN_DOUBLE(3.1415926535);}
PHP_FUNCTION(hello_bool)
{ RETURN_BOOL(1);}
PHP_FUNCTION(hello_null)
```

You'll also need to add prototypes for these functions alongside the prototypenfor hello_world() in the header file, php_hello.h, so that the build process takes place properly:

```
PHP_FUNCTION(hello_world);
PHP_FUNCTION(hello_long);
PHP_FUNCTION(hello_double);
PHP_FUNCTION(hello_bool);
PHP_FUNCTION(hello_null);
```

Since you made no changes to the config.m4 file, it's technically safe to skip the phpize and

./configure steps this time and jump straight to make. However, at this stage of the game I'm going to ask you to go through all three build steps again just to make sure you have a nice build. In additional, you should call make clean all rather than simply make in the last step, to ensure that all source files are rebuilt. Again, this isn't necessary because of the types of changes you've made so far, but better safe than confused. Once the module is built, you'll again copy it to your extension directory, replacing the old version.

At this point you could call the PHP interpreter again, passing it simple scripts to test out the functions you just added. In fact, why don't you do that now? I'll wait here…

Done? Good. If you used var_dump() rather than echo to view the output of each function then you probably noticed that hello_bool() returned true. That's what the 1 value RETURN_BOOL() represents. Just like in PHP scripts, an integer value of 0 equates to FALSE, while any other integer value equates to TRUE. Extension authors often use 1 as a matter convention, and you're encouraged to do the same, but don't feel locked into it. For added readability, the RETURN_TRUE and RETURN_FALSE macros are also available; here's hello_bool() again, this time using RETURN_TRUE:

```
PHP_FUNCTION(hello_bool)

{RETURN_TRUE;}
```

Note that no parentheses were used here. RETURN_TRUE and RETURN_FALSE are aberrations from the rest of the RETURN_*() macros in that way, so be sure not to get caught by this one!

You probably noticed in each of the code samples above that we didn't pass a zero or one value indicating whether or not the value should be copied. This is because no additional memory (beyond the variable container itself – we'll delve into this deeper in Part 2) needs to be allocated – or freed - for simple small scalars such as these.

There are an additional three return types: RESOURCE (as returned by mysql_connect(), fsockopen(), and ftp_connect() to name but a few), ARRAY (also known as a HASH), and OBJECT (as returned by the keyword new). We'll look at these in Part II of this series, when we cover variables in depth.

INI Settings

The Zend Engine provides two approaches for managing INI values. We'll take a look at the simpler approach for now, and explore the fuller, but more complex, approach later on, when

you've had a chance to work with global values.

Let's say you want to define a php.ini value for your extension, hello.greeting, which will hold the value used to say hello in your hello_world() function. You'll need to make a few additions to hello.c and php_hello.h while making a few key changes to the hello_module_entry structure. Start off by adding the followin prototypes near the userspace function prototypes in php_hello.h

```
PHP_MINIT_FUNCTION(hello);

PHP_MSHUTDOWN_FUNCTION(hello);

PHP_FUNCTION(hello_world);

PHP_FUNCTION(hello_long);

PHP_FUNCTION(hello_double);

PHP_FUNCTION(hello_bool);

PHP_FUNCTION(hello_null);
```

Now head over to hello.c and take out the current version of hello_module_entry, replacing it with the following listing:

```
zend_module_entry hello_module_entry = {

#if ZEND_MODULE_API_NO >= 20010901

    STANDARD_MODULE_HEADER,

#endif

    PHP_HELLO_WORLD_EXTNAME,

    hello_functions,

    PHP_MINIT(hello),

    PHP_MSHUTDOWN(hello),
```

```
    NULL,
    NULL,
    NULL,
#if ZEND_MODULE_API_NO >= 20010901
    PHP_HELLO_WORLD_VERSION,
#endif
    STANDARD_MODULE_PROPERTIES
};
PHP_INI_BEGIN()
PHP_INI_ENTRY("hello.greeting", "Hello World", PHP_INI_ALL, NULL)
PHP_INI_END()
PHP_MINIT_FUNCTION(hello)
{REGISTER_INI_ENTRIES();return SUCCESS;}
PHP_MSHUTDOWN_FUNCTION(hello)
{UNREGISTER_INI_ENTRIES();    return SUCCESS;}
```

Now, you just need to add an #include to the rest of the #includes at the top of hello.c to get the right headers for INI file support:

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif
#include "php.h"
```

#include "php_ini.h"

#include "php_hello.h"

Finally, you can modify your hello_world function to use the

INI value:

PHP_FUNCTION(hello_world)

{ RETURN_STRING(INI_STR("hello.greeting"), 1);}

Notice that you're copying the value returned by INI_STR(). This is because, as far as the PHP variable stack is concerned, this is a static string. In fact, if you tried to modify the string returned by this value, the PHP execution environment would become unstable and might even crash.

The first set of changes in this section introduced two methods you'll want to become very familiar with: MINIT, and MSHUTDOWN. As mentioned earlier, these methods are called during the initial startup of the SAPI layer and during its final shutdown, respectively. They are not called between or during requests. In this example you've used them to register the php.ini entries defined in your extension. Later in this series, you'll find how to use the MINIT and MSHUTDOWN functions to register resource, object, and stream handlers as well.

In your hello_world() function you used INI_STR() to retrieve the current value of the hello.greeting entry as a string. A host of other functions exist for retrieving values as longs, doubles, and Booleans as shown in the following table, along with a complementary ORIG counterpart which provides the value of the referenced INI setting as it was set in php.ini (before being altered by .htaccess or ini_set() statements).

Current Value Original Value Type

INI_STR(name) INI_ORIG_STR(name) char * (NULL terminated)

INI_INT(name) INI_ORIG_INT(name) signed long

INI_FLT(name) INI_ORIG_FLT(name) signed double

INI_BOOL(name) INI_ORIG_BOOL(name) zend_bool

The first parameter passed to PHP_INI_ENTRY() is a string containing the name of the entry to

be used in php.ini. In order to avoid namespace collisions, you should use the same conventions as with your functions; that is, prefix all values with the name of your extension, as you did with hello.greeting. As a matter of convention, a period is used to separate the extension name from the more descriptive part of the ini setting name.

The second parameter is the initial value, and is always given as a char* string regardless of whether it is a numerical value or not. This is due primarily to the fact that values in an .ini file are inherently textual – being a text file and all. Your use of INI_INT(), INI_FLT(), or INI_BOOL() later in your script will handle type conversions.

The third value you pass is an access mode modifier. This is a bitmask field which determines when and where this INI value should be modifiable. For some, such as register_globals, it simply doesn't make sense to allow the value to be changed from within a script using ini_set() because the setting only has meaning during request startup - before the script has had a chance to run. Others, such as allow_url_fopen, are administrative settings which you don't want to allow users on a shared hosting environment to change, either via ini_set() or through the use of .htaccess directives. A typical value for this parameter might be PHP_INI_ALL, indicating that the value may be changed anywhere. Then there's PHP_INI_SYSTEM|PHP_INI_PERDIR, indicating that the setting may be changed in the php.ini file, or via an Apache directive in a .htaccess file, but not through the use of ini_set(). Or there's PHP_INI_SYSTEM, meaning that the value may only be changed in the php.ini file and nowhere else.

We'll skip the fourth parameter for now and only mention that it allows the use of a callback method to be triggered whenever the ini setting is changed, such as with ini_set(). This allows an extension to perform more precise control over when a setting may be changed, or trigger a related action dependant on the new setting.

Global Values

Frequently, an extension will need to track a value through a particular request, keeping that value independent from other requests which may be occurring at the same time. In a non-threaded SAPI that might be simple: just declare a global variable in the source file and access it as needed. The trouble is, since PHP is designed to run on threaded web servers (such as Apache 2 and IIS), it needs to keep the global values used by one thread separate from the global values used by another. PHP greatly simplifies this by using the TSRM (Thread Safe Resource Management) abstraction layer, sometimes referred to as ZTS (Zend Thread Safety). In fact, by this point you've already used parts of TSRM and didn't even know it. (Don't search too hard just yet; as this series progresses you'll come to discover it's hiding everywhere.)

The first part of creating a thread safe global is, as with any global, declaring it. For the sake of this example, you'll declare one global value which will start out as a long with a value of 0. Each time the hello_long() function is called you'll increment this value and return

1. Add the following block of code to php_hello.h just after the

#define PHP_HELLO_H statement:

```
#ifdef ZTS

#include "TSRM.h"

#endif

ZEND_BEGIN_MODULE_GLOBALS(hello)

    long counter;

ZEND_END_MODULE_GLOBALS(hello)

#ifdef ZTS

#define HELLO_G(v) TSRMG(hello_globals_id, zend_hello_globals *, v)

#else

#define HELLO_G(v) (hello_globals.v)

#endif
```

You're also going to use the RINIT method this time around, so you

need to declare its prototype in the header:

```
PHP_MINIT_FUNCTION(hello);

PHP_MSHUTDOWN_FUNCTION(hello);

PHP_RINIT_FUNCTION(hello);
```

Now let’s go over to hello.c and add the following just after your include

block:

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif
#include "php.h"
#include "php_ini.h"
#include "php_hello.h"
ZEND_DECLARE_MODULE_GLOBALS(hello)
```

Change hello_module_entry by adding

PHP_RINIT(hello):

```
zend_module_entry hello_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_HELLO_WORLD_EXTNAME,
    hello_functions,
    PHP_MINIT(hello),
    PHP_MSHUTDOWN(hello),
    PHP_RINIT(hello),
```

```
    NULL,
    NULL,
#if ZEND_MODULE_API_NO >= 20010901
    PHP_HELLO_WORLD_VERSION,
#endif
    STANDARD_MODULE_PROPERTIES
```

And modify your MINIT function, along with the addition of another couple of functions, to handle initialization upon request startup:

```
static void
php_hello_init_globals(zend_hello_globals *hello_globals)
PHP_RINIT_FUNCTION(hello)
    HELLO_G(counter) = 0;
    return SUCCESS;
PHP_MINIT_FUNCTION(hello)
    ZEND_INIT_MODULE_GLOBALS(hello, php_hello_init_globals,
NULL);
    REGISTER_INI_ENTRIES();
    return SUCCESS;
```

Finally, you can modify the hello_long() function to use this value:

```
PHP_FUNCTION(hello_long)
    HELLO_G(counter)++;
```

```
RETURN_LONG(HELLO_G(counter));
```

In your additions to php_hello.h, you used a pair of macros - ZEND_BEGIN_MODULE_GLOBALS() and ZEND_END_MODULE_GLOBALS() - to create a struct named zend_hello_globals containing one variable of type long. You then conditionally defined HELLO_G() to either fetch this value from a thread pool, or just grab it from a global scope – if you're compiling for a non-threaded environment.

In hello.c you used the ZEND_DECLARE_MODULE_GLOBALS() macro to actually instantiate the zend_hello_globals struct either as a true global (if this is a non-thread-safe build), or as a member of this thread's resource pool. As extension authors, this distinction is one we don't need to worry about, as the Zend Engine takes care of the job for us. Finally, in MINIT, you used ZEND_INIT_MODULE_GLOBALS() to allocate a thread safe resource id – don't worry about what that is for now.

You may have noticed that php_hello_init_globals() doesn't actually do anything, yet we went to the trouble of declaring RINIT to initialize the counter to 0. Why?

The key lies in when the two functions are called. php_hello_init_globals() is only called when a new process or thread is started; however, each process can serve more than one request, so using this function to initialize our counter to 0 will only work for the first page request. Subsequent page requests to the same process will still have the old counter value stored here, and hence will not start counting from 0. To initialize the counter to 0 for every single page request, we implemented the RINIT function, which as you learned earlier is called prior to every page request. We included the php_hello_init_globals() function at this point because you'll be using it in a few moments, but also because passing a NULL to ZEND_INIT_MODULE_GLOBALS() for the init function will result in a segfault on non-threaded platforms.

INI Settings as Global Values

If you recall from earlier, a php.ini value declared with PHP_INI_ENTRY() is parsed as a string value and converted, as needed, to other formats with INI_INT(), INI_FLT(), and INI_BOOL(). For some settings, that represents a fair amount of unnecessary work duplication as the value is read over and over again during the course of a script's execution. Fortunately it's possible to instruct ZE to store the INI value in a particular data type, and only perform type conversions when its value is changed. Let's try that out by declaring another INI value, a Boolean this time, indicating whether the counter should increment, or decrement. Begin by changing the MODULE_GLOBALS block in php_hello.h to the following:

```
ZEND_BEGIN_MODULE_GLOBALS(hello)
    long counter;
    zend_bool direction;
ZEND_ENG_MODULE_GLOBALS(hello)
```

Next, declare the INI value itself by changing your

PHP_INI_BEGIN() block thus:

```
PHP_INI_BEGIN()
    PHP_INI_ENTRY("hello.greeting", "Hello World",
PHP_INI_ALL, NULL)
    STD_PHP_INI_ENTRY("hello.direction", "1", PHP_INI_ALL,
OnUpdateBool, direction, zend_hello_globals, hello_globals)
PHP_INI_END()
```

Now initialize the setting in the init_globals method with: static
void php_hello_init_globals(zend_hello_globals *hello_globals) {hello_globals->direction = 1;

And lastly, use the value of the ini setting in hello_long() to determine whether to increment or decrement:

```
PHP_FUNCTION(hello_long)
    if (HELLO_G(direction)) {
        HELLO_G(counter)++;
    } else {
        HELLO_G(counter)--;
```

```
    RETURN_LONG(HELLO_G(counter));
```

And that's it. The OnUpdateBool method you specified in the INI_ENTRY section will automatically convert any value provided in php.ini, .htaccess, or within a script via ini_set() to an appropriate TRUE/FALSE value which you can then access directly within a script. The last three parameters of STD_PHP_INI_ENTRY tell PHP which global variable to change, what the structure of our extension globals looks like, and the name of the global scope container where they're contained.

Sanity Check

By now our three files should look similar to the following listings. (A few items have been moved and grouped together, for the sake of readability.)

config.m4

```
PHP_ARG_ENABLE(hello, whether to enable Hello

World support,

[ --enable-hello   Enable Hello World support])

if test "$PHP_HELLO" = "yes"; then

  AC_DEFINE(HAVE_HELLO, 1, [Whether you have Hello World])

  PHP_NEW_EXTENSION(hello, hello.c, $ext_shared)
```

fiphp_hello.h

```
#ifndef PHP_HELLO_H

#define PHP_HELLO_H 1

#ifdef ZTS

#include "TSRM.h"

#endif
```

```
ZEND_BEGIN_MODULE_GLOBALS(hello)
    long counter;
    zend_bool direction;
ZEND_END_MODULE_GLOBALS(hello)
#ifdef ZTS
#define HELLO_G(v) TSRMG(hello_globals_id, zend_hello_globals *, v)
#else
#define HELLO_G(v) (hello_globals.v)
#endif
#define PHP_HELLO_WORLD_VERSION "1.0"
#define PHP_HELLO_WORLD_EXTNAME "hello"
PHP_MINIT_FUNCTION(hello);
PHP_MSHUTDOWN_FUNCTION(hello);
PHP_RINIT_FUNCTION(hello);
PHP_FUNCTION(hello_world);
PHP_FUNCTION(hello_long);
PHP_FUNCTION(hello_double);
PHP_FUNCTION(hello_bool);
PHP_FUNCTION(hello_null);
extern zend_module_entry hello_module_entry;
```

```
#define phpext_hello_ptr &hello_module_entry

#endif
```

hello.c

```
#ifdef HAVE_CONFIG_H

#include "config.h"

#endif

#include "php.h"

#include "php_ini.h"

#include "php_hello.h"

ZEND_DECLARE_MODULE_GLOBALS(hello)

static function_entry hello_functions[] = {

    PHP_FE(hello_world, NULL)

    PHP_FE(hello_long, NULL)

    PHP_FE(hello_double, NULL)

    PHP_FE(hello_bool, NULL)

    PHP_FE(hello_null, NULL)

    {NULL, NULL, NULL}

zend_module_entry hello_module_entry = {

#if ZEND_MODULE_API_NO >= 20010901

    STANDARD_MODULE_HEADER,
```

```
#endif
    PHP_HELLO_WORLD_EXTNAME,
    hello_functions,
    PHP_MINIT(hello),
    PHP_MSHUTDOWN(hello),
    PHP_RINIT(hello),
    NULL,
    NULL,
#if ZEND_MODULE_API_NO >= 20010901
    PHP_HELLO_WORLD_VERSION,
#endif
    STANDARD_MODULE_PROPERTIES
#ifdef COMPILE_DL_HELLO
ZEND_GET_MODULE(hello)
#endif
PHP_INI_BEGIN()
    PHP_INI_ENTRY("hello.greeting", "Hello World",
PHP_INI_ALL, NULL)
    STD_PHP_INI_ENTRY("hello.direction", "1", PHP_INI_ALL,
OnUpdateBool, direction, zend_hello_globals, hello_globals)
```

```
PHP_INI_END()

static void php_hello_init_globals(zend_hello_globals *hello_globals)
    hello_globals->direction = 1;
PHP_RINIT_FUNCTION(hello)
    HELLO_G(counter) = 0;
    return SUCCESS;
PHP_MINIT_FUNCTION(hello)
    ZEND_INIT_MODULE_GLOBALS(hello, php_hello_init_globals,
NULL);
    REGISTER_INI_ENTRIES();
    return SUCCESS;
PHP_MSHUTDOWN_FUNCTION(hello)
    UNREGISTER_INI_ENTRIES();
    return SUCCESS;
PHP_FUNCTION(hello_world)
    RETURN_STRING("Hello World", 1);
PHP_FUNCTION(hello_long)
    if (HELLO_G(direction)) {
        HELLO_G(counter)++;
    } else {
```

```
        HELLO_G(counter)--;

    RETURN_LONG(HELLO_G(counter));

PHP_FUNCTION(hello_double

    RETURN_DOUBLE(3.1415926535);

PHP_FUNCTION(hello_bool

    RETURN_BOOL(1);

PHP_FUNCTION(hello_null)

    RETURN_NULL();
```

What's Next?

In this tutorial we explored the structure of a simple PHP extension which exported functions, returned values, declared INI settings, and tracked its internal state during the course of a request.

In the next session we'll explore the internal structure of PHP variables, and how they're stored, tracked, and maintained within a script environment. We'll use zend_parse_parameters to receive parameters from a program when a function is called, and explore ways to return more complicated results, including the array, object, and resource types mentioned in this tutorial.