PHP at the Core: A Hacker's Guide to the Zend Engine

Preface

The Zend API has evolved considerably over time, as PHP has become a more robust and widespread language. With the introduction of PHP 5 came the Zend Engine 2 (ZE2). ZE2 came with an almost entirely new Object-Oriented Programming (OOP) model, as well as improvements in most aspects of the API. PHP 6, which is still under active development at the time of this writing, introduces the Zend Engine 3 (ZE3), which brings full Unicode support to the language.

Warning
This documentation is still under heavy development. The original Zend documentation is preserved in its entirety in the Zend Engine 1 section for those who need it before this documentation is completed.

This section of the manual is devoted to ZE2. While PHP 4.4 is still in widespread use, the differences in how extensions are written in ZE1 are small; a short reference to them is given in an appendix to this section. ZE3's API may yet change significantly, and is covered in another appendix. It will be more fully documented when PHP 6 enters a beta testing stage.

The documentation in this section is current as of PHP 5.2.5, the most recent stable release at the time of this writing. Notable differences in the minor PHP 5 releases (5.0 through 5.3) are given as appropriate.

The "counter" Extension - A Continuing Example

Preface

Throughout this Zend documentation, references are made to an example module in order to illustrate various concepts. The "counter" extension is this example, a fictional yet functional Zend module which strives to use as much of the Zend API as is reasonably possible. This short chapter describes the userland interface to the completed extension.

Note
"counter" serves no practical purpose whatsoever, as the functionality it provides is far more effectively implemented by appropriate userland code.

Installing/Configuring

Introduction

The "counter" extension provides any number of counters to PHP code using it which reset at times determined by the caller.

There are three interfaces to "counter": basic, extended, and objective. The basic interface provides a single counter controlled by INI settings and function calls. The extended interface provides an arbitrary number of named counter resources which may optionally persist beyond the lifetime of a single PHP request. The objective interface combines both the basic and extended interfaces into a Counter class.

Runtime Configuration

The behaviour of these functions is affected by settings in *php.ini*.

Counter configuration options

Name	Default	Changeable	Changelog
counter.reset_time	COUNTER_RESET_PER_REQUEST	PHP_INI_ALL	
counter.save_path	""	PHP_INI_ALL	
counter.initial_value	"0"	PHP_INI_ALL	

For further details and definitions of the PHP_INI_* constants, see the php.ini directives.

`counter.reset_time` integer
: *counter.reset_time* tells "counter" to reset the counter used by the basic interface. It may be any of the **COUNTER_RESET_*** constants (see below).

`counter.save_path` string
: Tells "counter" where to save data that has to persist between invocations of PHP (i.e. any counter that has COUNTER_RESET_NEVER or COUNTER_FLAG_SAVE). A file will be created at this path, which must be readable and writeable to whatever user PHP is running as.

`counter.initial_value` integer
: Sets the initial value of the counter used by the basic interface whenever it is reset.

Resource Types

The "counter" extension defines one resource type, a counter.

Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

COUNTER_FLAG_PERSIST (integer)
A counter with this flag will be created as a persistent resource.

COUNTER_FLAG_SAVE (integer)
A counter with this flag will be saved between invocations of PHP.

COUNTER_FLAG_NO_OVERWRITE (integer)
This flag causes counter_create() to avoid overwriting an existing named counter with a new one.

COUNTER_META_NAME (string)
Pass this constant to get the name of a counter resource or object.

COUNTER_META_IS_PERISTENT (string)
Pass this constant to determine whether a counter resource or object is persistent (has the **COUNTER_FLAG_PERSIST** flag).

COUNTER_RESET_NEVER (integer)
The counter will never be reset.

COUNTER_RESET_PER_LOAD (integer)
The counter will be reset on each invocation of PHP.

COUNTER_RESET_PER_REQUEST (integer)
The counter will be reset on each request.

Examples

Basic interface

The basic interface provides three simple functions, illustrated here:

Example #1 - "counter"'s basic interface

```
<?php
$starting_counter_value = counter_get();
counter_bump(1);
$second_counter_value = counter_get();
counter_reset();
$final_counter_value = counter_get();
printf("%3d %3d %3d", $starting_counter_value, $second_counter_value,
$final_counter_value);
?>
```

The above example will output:

```
0   1   0
```

The basic interface also provides a number of INI settings, documented below.

Extended interface

The extended interface provides a small suite of functions that allow the user to define an arbitrary number of named counters with unique settings. The basic interface can be used in parallel with the extended interface.

Example #2 - "counter"'s extended interface

```
<?php
function print_counter_info($counter)
{
   if (is_resource($counter)) {
       printf("Counter's name is '%s' and is%s persistent. Its current value
is %d.\n",
           counter_get_meta($counter, COUNTER_META_NAME),
           counter_get_meta($counter, COUNTER_META_IS_PERSISTENT) ? '' : '
not',
           counter_get_value($counter));
   } else {
     print "Not a valid counter!\n";
   }
}

if (($counter_one = counter_get_named("one")) === NULL) {
```

```
   $counter_one = counter_create("one", 0, COUNTER_FLAG_PERSIST);
}
counter_bump_value($counter_one, 2);
$counter_two = counter_create("two", 5);
$counter_three = counter_get_named("three");
$counter_four = counter_create("four", 2, COUNTER_FLAG_PERSIST |
COUNTER_FLAG_SAVE | COUNTER_FLAG_NO_OVERWRITE);
counter_bump_value($counter_four, 1);

print_counter_info($counter_one);
print_counter_info($counter_two);
print_counter_info($counter_three);
print_counter_info($counter_four);
?>
```

When run once, the above example outputs:

```
Counter's name is 'one' and is persistent. Its current value is 2.
Counter's name is 'two' and is not persistent. Its current value is 5.
Not a valid counter!
Counter's name is 'four' and is persistent. Its current value is 3.
```

If run a second time within the same instance of PHP, it outputs:

```
Counter's name is 'one' and is persistent. Its current value is 4.
Counter's name is 'two' and is not persistent. Its current value is 5.
Not a valid counter!
Counter's name is 'four' and is persistent. Its current value is 4.
```

If then run a third time *in a different instance of PHP*, it outputs:

```
Counter's name is 'one' and is persistent. Its current value is 2.
Counter's name is 'two' and is not persistent. Its current value is 5.
Not a valid counter!
Counter's name is 'four' and is persistent. Its current value is 5.
```

Objective interface

The objective interface provides an object-oriented way to access the extended interfaces. The following example shows how the above one would be implemented using the objective interface. The output of this example is exactly the same, except that instead of printing "Not a valid counter!", this will instead issue a PHP warning that the variable *$counter_three* is not an object. This example shows that it is possible to subclass the Counter class defined by the extension, as well as that the counter's value is maintained using an instance variable rather than method access.

Example #3 - "counter"'s objective interface

```
<?php
class MyCounter extends Counter
{
   public function printCounterInfo() {
       printf("Counter's name is '%s' and is%s persistent. Its current value
is %d.\n",
```

```
            $this->getMeta(COUNTER_META_NAME),
            $this->getMeta(COUNTER_META_IS_PERSISTENT) ? '' : ' not',
            $this->value);
   }
}

Counter::setCounterClass("MyCounter");
if (($counter_one = Counter::getNamed("one")) === NULL) {
   $counter_one = new Counter("one", 0, COUNTER_FLAG_PERSIST);
}
$counter_one->bumpValue(2); // we aren't allowed to "set" the value directly
$counter_two = new Counter("two", 5);
$counter_three = Counter::getNamed("three");
$counter_four = new Counter("four", 2, COUNTER_FLAG_PERSIST |
COUNTER_FLAG_SAVE | COUNTER_FLAG_NO_OVERWRITE);
$counter_four->bumpValue(1);

$counter_one->printCounterInfo();
$counter_two->printCounterInfo();
$counter_three->printCounterInfo();
$counter_four->printCounterInfo();
?>
```

The Counter class

Introduction

Represents a single counter object.

Class synopsis

Counter

Counter {

Counter::__construct (string `$name` [, integer `$initial_value` [, integer `$flags`]])

integer **Counter::getValue** (void)

void **Counter::bumpValue** (integer `$offset`)

void **Counter::resetValue** (void)

mixed **Counter::getMeta** (integer `$attribute`)

static Counter **Counter::getNamed** (string `$name`)

static void **Counter::setCounterClass** (string `$name`)
}

Counter::__construct

Counter::__construct -- Creates an instance of a Counter which maintains a single numeric value.

Description

Counter::__construct (string `$name` [, integer `$initial_value` [, integer `$flags`]])

Creates an instance of a Counter which maintains a single numeric value.

Parameters

name
: The new counter's name.

initial_value
: The initial value of the counter. Defaults to zero (0).

flags
: Flags for the new counter, chosen from the *COUNTER_FLAG_** constants.

Return Values

Returns a Counter object on success.

Errors/Exceptions

Counter::__construct() throws an Exception if something goes wrong.

Counter::getValue

Counter::getValue -- Get the current value of a counter.

Description

integer **Counter::getValue** (void)

Counter::getValue() returns the current value of a counter.

Return Values

Counter::getValue() returns an integer.

See Also

- Counter::bumpValue()
- Counter::resetValue()

Counter::bumpValue

Counter::bumpValue -- Change the current value of a counter.

Description

void **Counter::bumpValue** (integer `$offset`)

Counter::bumpValue() updates the current value of a counter.

Parameters

`offset`
: The amount by which to change the counter's value. Can be negative.

See Also

- Counter::getValue()
- Counter::resetValue()

Counter::resetValue

Counter::resetValue -- Reset the current value of a counter.

Description

void **Counter::resetValue** (void)

Counter::resetValue() resets the current value of a counter to its original initial value.

See Also

- Counter::getValue()
- Counter::bumpValue()

Counter::getMeta

Counter::getMeta -- Return a piece of metainformation about a counter.

Description

mixed **Counter::getMeta** (integer `$attribute`)

Counter::getMeta() returns metainformation about a counter.

Parameters

`attribute`
: The metainformation to retrieve.

Return Values

Counter::getMeta() returns values of varying types based on which metainformation was requested.

Counter::getNamed

Counter::getNamed -- Retrieve an existing named counter.

Description

static Counter **Counter::getNamed** (string `$name`)

Counter::getNamed() returns an existing counter by name if that name exists, or **NULL** otherwise. This is a static function.

Parameters

`name`
: The counter name to search for.

Return Values

Counter::getNamed() returns an existing counter by name if that name exists, or **NULL** otherwise.

Counter::setCounterClass

Counter::setCounterClass -- Set the class returned by **Counter::getNamed**.

Description

static void **Counter::setCounterClass** (string `$name`)

Counter::setCounterClass() changes the class of objects returned by Counter::getNamed() . The class being set must not have a public constructor and must be a subclass of Counter. If these conditions are not met, a fatal error is raised. This is a static function.

Parameters

`name`
: The name of the class to use.

The basic interface

counter_get

counter_get -- Get the current value of the basic counter.

Description

integer **counter_get** (void)

counter_get() returns the current value of the basic interface's counter.

Return Values

counter_get() returns an integer.

See Also

- counter_bump()
- counter_reset()

counter_bump

counter_bump -- Update the current value of the basic counter.

Description

void **counter_bump** (integer `$offset`)

counter_bump() updates the current value of the basic interface's counter.

Parameters

`offset`
: The amount by which to change the counter's value. Can be negative.

See Also

- counter_get()
- counter_reset()

counter_reset

counter_reset -- Reset the current value of the basic counter.

Description

void **counter_reset** (void)

counter_reset() resets the current value of the basic interface's counter to its original initial value.

See Also

- counter_get()
- counter_bump()

The extended interface

counter_create

counter_create -- Creates a counter which maintains a single numeric value.

Description

resource **counter_create** (string `$name` [, integer `$initial_value` [, integer `$flags`]])

Creates a counter which maintains a single numeric value.

Parameters

name
: The new counter's name.

initial_value
: The initial value of the counter. Defaults to zero (0).

flags
: Flags for the new counter, chosen from the *COUNTER_FLAG_** constants.

Return Values

Returns a counter resource.

counter_get_value

counter_get_value -- Get the current value of a counter resource.

Description

integer **counter_get_value** (resource `$counter`)

counter_get_value() returns the current value of a counter resource.

Parameters

`counter`
: The counter resource to operate on.

Return Values

counter_get_value() returns an integer.

See Also

- counter_bump_value()
- counter_reset_value()

counter_bump_value

counter_bump_value -- Change the current value of a counter resource.

Description

void **counter_bump_value** (resource `$counter`, integer `$offset`)

counter_bump_value() updates the current value of a counter resource.

Parameters

`counter`
: The counter resource to operate on.

`offset`
: The amount by which to change the counter's value. Can be negative.

See Also

- counter_get_value()
- counter_reset_value()

counter_reset_value

counter_reset_value -- Reset the current value of a counter resource.

Description

void **counter_reset_value** (resource `$counter`)

counter_reset_value() resets the current value of a counter resource to its original initial value.

Parameters

`counter`
: The counter resource to operate on.

See Also

- counter_get_value()
- counter_bump_value()

counter_get_meta

counter_get_meta -- Return a piece of metainformation about a counter resource.

Description

mixed **counter_get_meta** (resource `$counter`, integer `$attribute`)

counter_get_meta() returns metainformation about a counter resource.

Parameters

`counter`
: The counter resource to operate on.

`attribute`
: The metainformation to retrieve.

Return Values

counter_get_meta() returns values of varying types based on which metainformation was requested.

counter_get_named

counter_get_named -- Retrieve an existing named counter as a resource.

Description

resource **Counter::getNamed** (string `$name`)

counter_get_named() returns an existing counter by name if that name exists, or **NULL** otherwise.

Parameters

name
The counter name to search for.

Return Values

counter_get_name() returns an existing counter by name if that name exists, or **NULL** otherwise.

The PHP 5 build system

With all the functionality and flexibility available in PHP 5, it is no surprise that it consists of several thousand files and over one million lines of source code. Equally unsurprising is the necessity of a build system to manage so much data. This section describes how to set PHP up for extension development, the layout of an extension within the PHP source tree, and how to interface your extension with the build system.

Building PHP for extension development

In a typical PHP installation, the need for high performance almost always results in optimization at the cost of debugging facilities. This is a reasonable tradeoff for production use, but when developing an extension it falls short. What we need is a build of PHP which will give us some hints what has gone wrong when something does.

The Zend Engine provides a memory manager which is capable of tracking memory leaks in extensions and providing detailed debugging information. This tracking is disabled by default, as is thread-safety. To turn them on, pass the *--enable-debug* and *--enable-maintainer-zts* options to *configure*, along with whatever options you typically use. For instructions on building PHP from source, see the instructions at General Installation Considerations. A typical *configure* line might look like this:

```
$ ./configure --prefix=/where/to/install/php --enable-debug
--enable-maintainer-zts --enable-cgi --enable-cli --with-mysql=/path/to/mysql
```

The ext_skel script

A Zend extension is composed of several files common to all extensions. As the details of many of those files are similar from extension to extension, it can be laborous to duplicate the content for each one. Fortunately, there is a script which can do all of the initial setup for you. It's called *ext_skel*, and it's been distributed with PHP since 4.0.

Running *ext_skel* with no parameters produces this output in PHP 5.2.2:

```
php-5.2.2/ext$ ./ext_skel
./ext_skel --extname=module [--proto=file] [--stubs=file] [--xml[=file]]
           [--skel=dir] [--full-xml] [--no-help]

 --extname=module   module is the name of your extension
 --proto=file       file contains prototypes of functions to create
 --stubs=file       generate only function stubs in file
 --xml              generate xml documentation to be added to phpdoc-cvs
 --skel=dir         path to the skeleton directory
 --full-xml         generate xml documentation for a self-contained extension
                    (not yet implemented)
 --no-help          don't try to be nice and create comments in the code
                    and helper functions to test if the module compiled
```

Generally, when developing a new extension the only parameters you will be interested in are *--extname* and *--no-help*. Unless you are already experienced with the structure of an extension, you will *not* want to use *--no-help*; specifying it causes *ext_skel* to leave out

many helpful comments in the files it generates.

This leaves you with *--extname*, which tells *ext_skel* what the name of your extension is. This "name" is an all-lowercase identifier containing only letters and underscores which is unique among everything in the *ext/* folder of your PHP distribution.

The *--proto* option is intended to allow the developer to specify a header file from which a set of PHP functions will be created, ostensibly for the purpose of developing an extension based on a library, but it often functions poorly with most modern header files. A test run on the *zlib.h* header resulted in a very large number of empty and nonsense prototypes in the *ext_skel* output files. The *--xml* and *--full-xml* options are entirely nonfunctional thus far. The *--skel* option can be used to specify a modified set of skeleton files to work from, a topic which is beyond the scope of this section.

Talking to the UNIX build system: config.m4

The *config.m4* file for an extension tells the UNIX build system what *configure* options your extension supports, what external libraries and includes you require, and what source files are to be compiled as part of it. A reference to all the commonly used autoconf macros, both PHP-specific and those built into autoconf, is given in the Zend Engine 2 API reference section.

Tip

When developing a PHP extension, it is *strongly* recommended that *autoconf* version 2.13 be installed, despite the newer releases which are available. Version 2.13 is recognized as a common denominator of *autoconf* availability, usability, and user base. Using later versions will sometimes produce cosmetic differences from the expected output of *configure*.

Example #4 - An example config.m4 file

```
dnl $Id$
dnl config.m4 for extension examplePHP_ARG_WITH(example, for example
support,
[  --with-example[=FILE]       Include example support. File is the optional
path to example-config])
PHP_ARG_ENABLE(example-debug, whether to enable debugging support in
example,
[  --enable-example-debug        example: Enable debugging support in
example], no, no)
PHP_ARG_WITH(example-extra, for extra libraries for example,
[  --with-example-extra=DIR      example: Location of extra libraries for
example], no, no)

dnl Check whether the extension is enabled at all
if test "$PHP_EXAMPLE" != "no"; then

 dnl Check for example-config. First try any path that was given to us, then
```

```
look in $PATH
 AC_MSG_CHECKING([for example-config])
 EXAMPLE_CONFIG="example-config"
 if test "$PHP_EXAMPLE" != "yes"; then
   EXAMPLE_PATH=$PHP_EXAMPLE
 else
   EXAMPLE_PATH=`$php_shtool path $EXAMPLE_CONFIG`
 fi

 dnl If a usable example-config was found, use it
 if test -f "$EXAMPLE_PATH" && test -x "$EXAMPLE_PATH" && $EXAMPLE_PATH
--version > /dev/null 2>&1; then
   AC_MSG_RESULT([$EXAMPLE_PATH])
   EXAMPLE_LIB_NAME=`$EXAMPLE_PATH --libname`
   EXAMPLE_INCDIRS=`$EXAMPLE_PATH --incdirs`
   EXAMPLE_LIBS=`$EXAMPLE_PATH --libs`

   dnl Check that the library works properly
   PHP_CHECK_LIBRARY($EXAMPLE_LIB_NAME, example_critical_function,
   [
     dnl Add the necessary include dirs
     PHP_EVAL_INCLINE($EXAMPLE_INCDIRS)
     dnl Add the necessary libraries and library dirs
     PHP_EVAL_LIBLINE($EXAMPLE_LIBS, EXAMPLE_SHARED_LIBADD)
   ],[
     dnl Bail out
     AC_MSG_ERROR([example library not found. Check config.log for more
information.])
   ],[$EXAMPLE_LIBS]
   )
 else
   dnl No usable example-config, bail
   AC_MSG_RESULT([not found])
   AC_MSG_ERROR([Please check your example installation.])
 fi

 dnl Check whether to enable debugging
 if test "$PHP_EXAMPLE_DEBUG" != "no"; then
   dnl Yes, so set the C macro
   AC_DEFINE(USE_EXAMPLE_DEBUG,1,[Include debugging support in example])
 fi

 dnl Check for the extra support
 if test "$PHP_EXAMPLE_EXTRA" != "no"; then
   if test "$PHP_EXAMPLE_EXTRA" == "yes"; then
     AC_MSG_ERROR([You must specify a path when using --with-example-extra])
   fi

   PHP_CHECK_LIBRARY(example-extra, example_critical_extra_function,
   [
     dnl Add the neccessary paths
     PHP_ADD_INCLUDE($PHP_EXAMPLE_EXTRA/include)
     PHP_ADD_LIBRARY_WITH_PATH(example-extra, $PHP_EXAMPLE_EXTRA/lib,
EXAMPLE_SHARED_LIBADD)
     AC_DEFINE(HAVE_EXAMPLEEXTRALIB,1,[Whether example-extra support is
present and requested])
     EXAMPLE_SOURCES="$EXAMPLE_SOURCES example_extra.c"
   ],[
     AC_MSG_ERROR([example-extra lib not found. See config.log for more
information.])
```

```
   ],[-L$PHP_EXAMPLE_EXTRA/lib]
   )
 fi

 dnl Finally, tell the build system about the extension and what files are
needed
 PHP_NEW_EXTENSION(example, example.c $EXAMPLE_SOURCES, $ext_shared)
 PHP_SUBST(EXAMPLE_SHARED_LIBADD)
fi
```

A short introduction to autoconf syntax

config.m4 files are written using the GNU *autoconf* syntax. It can be described in a nutshell as shell scripting augmented by a powerful macro language. Comments are delimited by the string *dnl*, and strings are quoted using left and right brackets (e.g. *[* and *]*). Quoting of strings can be nested as many times as needed. A full reference to the syntax can be found in the *autoconf* manual at.

PHP_ARG_*: Giving users the option

The very first thing seen in the example *config.m4* above, aside from a couple of comments, are three lines using **PHP_ARG_WITH()** and **PHP_ARG_ENABLE()**. These provide *configure* with the options and help text seen when running*./configure --help*. As the names suggest, the difference between the two is whether they create a *--with-** option or an *--enable-** option. Every extension should provide at least one or the other with the extension name, so that users can choose whether or not to build the extension into PHP. By convention, **PHP_ARG_WITH()** is used for an option which takes a parameter, such as the location of a library or program required by an extension, while **PHP_ARG_ENABLE()** is used for an option which represents a simple flag.

Example #5 - Sample configure output

```
$ ./configure --help
...
 --with-example[=FILE]       Include example support. FILE is the optional
path to example-config
 --enable-example-debug        example: Enable debugging support in example
 --with-example-extra=DIR      example: Location of extra libraries for
example
...

$ ./configure --with-example=/some/library/path/example-config
--disable-example-debug --with-example-extra=/another/library/path
...
checking for example support... yes
checking whether to enable debugging support in example... no
checking for extra libraries for example... /another/library/path
...
```

Note
Regardless of the order in which options are specified on the command line when *configure* is called, the checks will be run in the order they are specified in *config.m4*.

Processing the user's choices

Now that *config.m4* can provide the user with some choices of what to do, it's time to act upon those choices. In the example above, the obvious default for all three options, if any of them are unspecified, is "no". As a matter of convention, it is best to use this as the default for the option which enables the extension, as it will be overridden by *phpize* for extensions built separately, and should not clutter the extension space by default when being built into PHP. The code to process the three options is by far the most complicated.

Handling the --with-example[=FILE] option

The first check made of the *--with-example[=FILE]* option is whether it was set at all. As this option controls the inclusion of the entire extension, if it was unspecified, given in the negative form (*--without-example*), or given the value "no", nothing else is done at all. In the example above, it is specified with the value */some/library/path/example-config*, so the first test succeeds.

Next, the code calls **AC_MSG_CHECKING()**, an *autoconf* macro which outputs a standard "checking for something" line, and checks whether the user gave an explicit path to the fictional *example-config*. In this example, *PHP_EXAMPLE* got the value */some/library/path/example-config*, which is now copied into the EXAMPLE_PATH variable. Had the user specified only *--with-example*, the code would have executed *$php_shtool path $EXAMPLE_CONFIG*, which would try to guess the location of *example-config* using the user's current *PATH*. Either way, the next step is to check whether the chosen *EXAMPLE_PATH* is a regular file, is executable, and can be run successfully. If so, **AC_MSG_RESULT()** is called, which completes the output line started by **AC_MSG_CHECKING()**. Otherwise, **AC_MSG_ERROR()** is called, which prints the given message and halts *configure* immediately.

The code now determines some site-specific configuration information by running *example-config* several times. The next call is to **PHP_CHECK_LIBRARY()**, a macro provided by the PHP buildsystem as a wrapper around *autoconf* 's **AC_CHECK_LIB()**. **PHP_CHECK_LIBRARY()** attempts to compile, link, and run a program which calls the symbol specified by the second parameter in the library specified by the first, using the string given in the fifth as extra linker options. If the attempt succeeds, the script given in the third parameter is run. This script tells the PHP buildsystem to extract include paths, library paths, and library names from the raw option strings *example-config* provided. If the attempt fails, the script in the fourth parameter is run instead. In this case, **AC_MSG_ERROR()** is called to stop processing.

Handling the --enable-example-debug option

Processing the *--enable-example-debug* is much simpler. A simple check for its truth value is performed. If that check succeeds, **AC_DEFINE()** is called to make the C macro *USE_EXAMPLE_DEBUG* available to the source of the extension. The third parameter is a comment string for *config.h*; it is safe to leave this empty, and often is.

Handling the --with-example-extra=DIR option

For the sake of this example, the fictional "extra" functionality requested by the *--with-example-extra=DIR* option does not share the fictional *example-config* program, nor does it have any default paths to search. Therefore, the user is required to provide the installation prefix of the necessary library. This setup is somewhat unlikely in a real-world extension, but is considered illustrative.

The code begins in a now-familiar way by checking the truth value of *PHP_EXAMPLE_EXTRA*. If a negative form was provided, no further processing is done; the user did not request extra functionality. If a positive form was provided without a parameter, **AC_MSG_ERROR()** is called to halt processing. The next step is another invocation of **PHP_CHECK_LIBRARY()**. This time, since there is no set of predefined compiler options provided, **PHP_ADD_INCLUDE()** and **PHP_ADD_LIBRARY_WITH_PATH()** are used to construct the necessary include paths, library paths, and library flags for the extra functionality. **AC_DEFINE()** is also called to indicate to the code that the extra functionality was both requested and available, and a variable is set to tell later code that there are extra source files to build. If the check fails, the familiar **AC_MSG_ERROR()** is called. A different way to handle the failure would have been to call **AC_MSG_WARNING()** instead, e.g.:

```
AC_MSG_WARNING([example-extra lib not found. example will be built without extra
functionality.])
```

In this case, *configure* would print a warning message rather than an error, and continue processing. Which way such failures are handled is a design decision left to the extension developer.

Telling the buildsystem what was decided

With all the necessary includes and libraries specified, with all the options processed and macros defined, one more thing remains to be done: The build system must be told to build the extension itself, and which files are to be used for that. To do this, the **PHP_NEW_EXTENSION()** macro is called. The first parameter is the name of the extension, which is the same as the name of the directory containing it. The second parameter is the list of all source files which are part of the extension. See **PHP_ADD_BUILD_DIR()** for information about adding source files in subdirectories to the build process. The third parameter should always be *$ext_shared*, a value which was determined by *configure* when **PHP_ARG_WITH()** was called for *--with-example[=FILE]*. The fourth parameter specifies a "SAPI class", and is only useful for extensions which require the CGI or CLI SAPIs specifically. It should be left empty in all other cases. The fifth parameter specifies a list of flags to be added to *CFLAGS* while building the extension; the sixth is a boolean value which, if "yes", will force the entire extension to be

built using *$CXX* instead of *$CC*. All parameters after the third are optional. Finally, **PHP_SUBST()** is called to enable shared builds of the extension. See Extension FAQs for more information on disabling support for building an extension in shared mode.

The counter extension's config.m4 file

The counter extension previously documented has a much simpler *config.m4* file than that described above, as it doesn't make use of many buildsystem features. This is a preferred method of operation for any extension that doesn't use an external or bundled library.

Example #6 - counter's config.m4 file

```
dnl$Id$
dnl config.m4 for extension counter

PHP_ARG_ENABLE(counter, for counter support,
[  --enable-counter            Include counter support])

dnl Check whether the extension is enabled at all
if test "$PHP_COUNTER" != "no"; then
 dnl Finally, tell the build system about the extension and what files are
needed
 PHP_NEW_EXTENSION(counter, counter.c counter_util.c, $ext_shared)
 PHP_SUBST(COUNTER_SHARED_LIBADD)
fi
```

Talking to the Windows build system: config.w32

An extension's *config.w32* file is similar in usage to the *config.m4* file, with two critical differences: first, it is used for Windows builds, and second, it is written in JavaScript. This section makes no attempt to cover JavaScript syntax. For the moment, this section is incomplete in lieu of a Win32 testbed, and an experimental-only port of the example *config.m4* is the only example provided.

Example #7 - An example config.w32 file

```
// $Id$
// vim:ft=javascriptARG_WITH("example", "for example support", "no");
ARG_ENABLE("example-debug", "for debugging support in example", "no")
ARG_WITH("example-extra", "for extra functionality in example", "no")
if (PHP_EXAMPLE != "no") {
   if (CHECK_LIB("libexample.lib", "example", PHP_EXAMPLE) &&
       CHECK_HEADER_ADD_INCLUDE("example.h", "CFLAGS_EXAMPLE", PHP_EXAMPLE +
"\\include")) {

       if (PHP_EXAMPLE_DEBUG != "no") {
           AC_DEFINE('USE_EXAMPLE_DEBUG', 1, 'Debug support in example');
       }

       if (PHP_EXAMPLE_EXTRA != "no" &&
```

```
            CHECK_LIB("libexample-extra.lib", "example", PHP_EXAMPLE) &&
            CHECK_HEADER_ADD_INCLUDE("example-extra.h", "CFLAGS_EXAMPLE",
PHP_EXAMPLE + ";" + PHP_PHP_BUILD + "\\include") {
            AC_DEFINE('HAVE_EXAMPLEEXTRA', 1, 'Extra functionality in
example');
            HAVE_EXTRA = 1;
        } else {
            WARNING( "extra example functionality not enabled, lib not found"
);
        }

        EXTENSION("example", "example.c");
        if (HAVE_EXTRA == 1) {
            ADD_SOURCES("example-extra.c");
        }
    } else {
        WARNING( "example not enabled; libraries not found" );
    }
}
```

The counter extension's config.w32 file

The counter extension previously documented has a much simpler *config.w32* file than that described above, as it doesn't make use of many buildsystem features.

Example #8 - counter's config.w32 file

```
// $Id$
// vim:ft=javascriptARG_ENABLE("counter", "for counter support", "no");
if (PHP_COUNTER != "no") {
 EXTENSION("counter", "counter.c");
 ADD_SOURCE("counter-util.c");
}
```

Extension structure

Many extension-writing guides focus on simple examples first and ignore the requirements of more complex implementations until later. Often such guides must repeat themselves over and over in order to describe these new features. This section describes extension structure from the perspective of a mature, practical implementation, in order to prepare users for needs and issues they will almost always encounter in the process of extension development.

Files which make up an extension

Whether created by hand, using *ext_skel*, or by an alternate extension generator, such as » CodeGen, all extensions will have at least four files:

config.m4
: UNIX build system configuration (see Talking to the UNIX build system: config.m4)

config.w32
: Windows buildsystem configuration (see Talking to the Windows build system: config.w32)

php_counter.h
: When building an extension as static module into the PHP binary the build system expects a header file with *php_* prepended to the extension name which includes a declaration for a pointer to the extension's module structure. This file usually contains additional macros, prototypes, and globals, just like any header.

counter.c
: Main extension source file. By convention, the name of this file is the extension name, but this is not a requirement. This file contains the module structure declaration, INI entries, management functions, userspace functions, and other requirements of an extension.

The buildsystem files are discussed elsewhere; this section concentrates on the rest. These four files make up the bare minimum for an extension, which may also contain any number of headers, source files, unit tests, and other support files. The list of files in the counter extension might look like this:

Example #9 - Files in the counter extension, in no particular order

```
ext/
counter/
 .cvsignore
 config.m4
 config.w32
 counter_util.h
 counter_util.c
 php_counter.h
 counter.c
```

```
package.xml
CREDITS
tests/
 critical_function_001.phpt
 critical_function_002.phpt
 optional_function_001.phpt
 optional_function_002.phpt
```

Non-source files

The *.cvsignore* file is used for extensions which are checked into one of the PHP *CVS* repositories (usually » PECL); the one generated by *ext_skel* contains:

```
.deps
*.lo
*.la
```

These lines tell *CVS* to ignore interim files generated by the PHP buildsystem. This is only a convenience, and can be omitted completely without ill effect.

The *CREDITS* file lists the contributors and/or maintainers of the extension in plain text format. The main purpose of this file is generating the credits information for bundled extensions as used by phpcredits(). By convention the first line of the file should hold the name of the extension, the second a comma separated list of contributors. The contributors are usually ordered by the chronological order of their contributions. In a » PECL package, this information is already maintained in *package.xml*, for example. This is another file which can be omitted without ill effect.

The *package.xml* file is specific to » PECL -based extensions; it is a metainformation file which gives details about an extension's dependencies, authors, installation requirements, and other tidbits. In an extension not being hosted in » PECL, this file is extraneous.

Basic constructs

C is a very low-level language by modern definitions. This means that it has no built-in support for many features that PHP takes for granted, such as reflection, dynamic module loading, bounds checking, threadsafe data management and various useful data structures including linked lists and hash tables. At the same time, C is a common denominator of language support and functionality. Given enough work, none of these concepts are impossible; the Zend Engine uses them all.

A lot of effort has gone into making the Zend API both extensible and understandable, but C forces certain necessary declarations upon any extension that to an inexperienced eye seem redundant or plain unnecessary. All of those constructs, detailed in this section, are "write once and forget" in Zend Engine 2 and 3. Here are some excerpts from the pregenerated *php_counter.h* and *counter.c* files created by PHP 5.3's *ext_skel*, showing the pregenerated declarations:

Note

The astute reader will notice that there are several delcarations in the real files that aren't shown here. Those declaractions are specific to various Zend subsystems and are discussed elsewhere as appropriate.

```
extern zend_module_entry counter_module_entry;
#define phpext_counter_ptr &counter_module_entry

#ifdef PHP_WIN32
# define PHP_COUNTER_API __declspec(dllexport)
#elif defined(__GNUC__) && __GNUC__ >= 4
# define PHP_COUNTER_API __attribute__ ((visibility("default")))
#else
# define PHP_COUNTER_API
#endif

#ifdef ZTS
#include "TSRM.h"
#endif

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

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "php_counter.h"

/* ... */

#ifdef COMPILE_DL_COUNTER
ZEND_GET_MODULE(counter)
#endif
```

- The lines concerning *counter_module_entry* declare a global variable, and a macroed pointer to it, which contains the *zend_module_entry* for the extension. Despite the later discussion regarding the drawbacks of "true" globals, this usage is intentional; Zend takes precautions to avoid misusing this variable.
- **PHP_COUNTER_API** is declared for use by non-PHP functions the module intends to export for the use of other modules. The counter extension doesn't declare any of these, and in the final version of the header file, this macro has been removed. The **PHPAPI** macro is declared identically elsewhere and is used by the standard extension to make the phpinfo() utility functions available to other extensions.
- The include of *TSRM.h* is skipped if PHP, or the extension, isn't being compiled with thread-safety, since in that case TSRM isn't used.
- A standard list of includes, especially the extension's own *php_counter.h*, is given. *config.h* gives the extension access to determinations made by *configure*. *php.h* is the gateway to the entire PHP and Zend APIs. *php_ini.h* adds the APIs for runtime configuration (INI) entries. Not all extensions will use this. Finally, *ext/standard/info.h*

imports the aforementioned phpinfo() utility API.

- **COMPILE_DL_COUNTER** will only be defined by *configure* if the counter extension is both enabled and wants to be built as a dynamically loadable module instead of being statically linked into PHP. **ZEND_GET_MODULE** defines a tiny function which Zend can use to get the extension's *zend_module_entry* at runtime.

> **Note**
>
> The astute reader who has peeked into *main/php_config.h* after trying to build with the counter module enabled statically may have noticed that there is also a **HAVE_COUNTER** constant defined that the source code doesn't check for. There's a simple reason this check isn't done: It's unnecessary. If the extension isn't enabled, the source file will never be compiled.

The zend_module structure

The main source file of a PHP extension contains several new constructs for a C programmer. The most important of these, the one touched first when starting a new extension, is the *zend_module* structure. This structure contains a wealth of information that tells the Zend Engine about the extension's dependencies, version, callbacks, and other critical data. The structure has mutated considerably over time; this section will focus on the structure as it has appeared since PHP 5.2, and will identify the very few parts which have changed in PHP 5.3.

The *zend_module* declaration from *counter.c* looks like this before any code has been written. The example file was generated by *ext_skel --extname=counter*, with some obsolete constructs removed:

Example #10 - zend_module declaration in the counter extension

```
/* {{{ counter_module_entry
*/
zend_module_entry counter_module_entry = {
   STANDARD_MODULE_HEADER,
   "counter",
   counter_functions,
   PHP_MINIT(counter),
   PHP_MSHUTDOWN(counter),
   PHP_RINIT(counter),        /* Replace with NULL if there's nothing to do
at request start */
   PHP_RSHUTDOWN(counter),    /* Replace with NULL if there's nothing to do
at request end */
   PHP_MINFO(counter),
   "0.1", /* Replace with version number for your extension */
   STANDARD_MODULE_PROPERTIES
};
/* }}} */
```

This may look a bit daunting at first glance, but most of it is very simple to understand. Here's the declaration of *zend_module* from *zend_modules.h* in PHP 5.3:

Example #11 - zend_module definition in PHP 5.3

```
struct _zend_module_entry {
   unsigned short size;
   unsigned int zend_api;
   unsigned char zend_debug;
   unsigned char zts;
   const struct _zend_ini_entry *ini_entry;
   const struct _zend_module_dep *deps;
   const char *name;
   const struct _zend_function_entry *functions;
   int (*module_startup_func)(INIT_FUNC_ARGS);
   int (*module_shutdown_func)(SHUTDOWN_FUNC_ARGS);
   int (*request_startup_func)(INIT_FUNC_ARGS);
   int (*request_shutdown_func)(SHUTDOWN_FUNC_ARGS);
   void (*info_func)(ZEND_MODULE_INFO_FUNC_ARGS);
   const char *version;
   size_t globals_size;
#ifdef ZTS
   ts_rsrc_id* globals_id_ptr;
#else
   void* globals_ptr;
#endif
   void (*globals_ctor)(void *global TSRMLS_DC);
   void (*globals_dtor)(void *global TSRMLS_DC);
   int (*post_deactivate_func)(void);
   int module_started;
   unsigned char type;
   void *handle;
   int module_number;
};
```

Many of these fields will never be touched by an extension writer. There are a number of standard macros that set them to their proper values automatically. The macro **STANDARD_MODULE_HEADER** fills in everything up to the *deps* field. Alternatively, the **STANDARD_MODULE_HEADER_EX** will leave the *deps* field empty for the developer's use. The developer is always responsible for everything from *name* to *version*. After that, the **STANDARD_MODULE_PROPERTIES** macro will fill in the rest of the structure, or the **STANDARD_MODULE_PROPERTIES_EX** macro can be used to leave the extension globals and post-deactivation function fields unfilled. Most modern extensions will make use of module globals.

Note

This table gives the values that each field would have if the developer were to fill in the structure entirely by hand, without recourse to any of the shortcut macros. *This is not recommended.* The "correct" values for many fields may change. Use the macros whenever possible.

Module structure field values

Field	Value	Description
size [1] [2] [3]	`sizeof(zend_module_entry)`	The size in bytes of the structure.
zend_api [1] [2] [3]	**ZEND_MODULE_API_NO**	The version of the Zend API this module was compiled against.
zend_debug [1] [2] [3]	**ZEND_DEBUG**	A flag indicating whether the module was compiled with debugging turned on.
zts [1] [2] [3]	**USING_ZTS**	A flag indicating whether the module was compiled with ZTS (TSRM) enabled (see Memory management).
ini_entry [1] [3]	**NULL**	This pointer is used internally by Zend to keep a non-local reference to any INI entries declared for the module.
deps [3]	**NULL**	A pointer to a list of dependencies for the module.
name	"mymodule"	The name of the module. This is the short name, such as "spl" or "standard".
functions	mymodule_functions	A pointer to the module's function table, which Zend uses to expose functions in the module to user space.
module_startup_func	PHP_MINIT(mymodule)	A callback function that Zend will call the first time a module is loaded into a particular instance of PHP.
module_shutdown_func	PHP_MSHUTDOWN(mymodule)	A callback function that Zend will call the when a module is unloaded from a particular instance of PHP, typically during final shutdown.
request_startup_func	PHP_RINIT(mymodule)	A callback function that Zend will call at the beginning of each request.

request_shutdown_func	PHP_RSHUTDOWN(mymodule)	A callback function that Zend will call at the end of each request.
info_func	PHP_MINFO(mymodule)	A callback function that Zend will call when the phpinfo() function is called.
version	**NO_VERSION_YET**	A string giving the version of the module, as specified by the module developer. It is recommended that the version number be either in the format expected by version_compare() (e.g. "1.0.5-dev"), or a CVS or SVN revision number (e.g. "Rev").
globals_size [1] [4] [5] [6]	sizeof(zend_mymodule_globals)	The size of the data structure containing the module's globals, if any.
globals_id_ptr [1] [4] [5] [6] [7]	&mymodule_globals_id	Only one of these two fields will exist, depending upon whether the **USING_ZTS** constant is **TRUE**. The former is an index into TSRM's allocation table for the module's globals, and the latter is a pointer directly to the globals.
globals_ptr [1] [4] [5] [6] [8]	&mymodule_globals	
globals_ctor [4] [5] [6]	PHP_GINIT(mymodule)	This funtion is called to initialize a module's globals *before* any *module_startup_func*.
globals_dtor [4] [5] [6]	PHP_GSHUTDOWN(mymodule)	This funtion is called to deallocate a module's globals *after* any *module_shutdown_func*.
post_deactivate_func [4]	ZEND_MODULE_POST_ZEND_DEACTIVATE_N(mymodule)	This function is called by Zend after request shutdown. It is rarely used.
module_started [1] [9] [4]	0	These fields are used for Zend's internal tracking information.

type [1] [9] [4]	0	
handle [1] [9] [4]	**NULL**	
module_number [1] [9] [4]	0	

[1] This field is not intended for use by module developers.
[2] This field is filled in by **STANDARD_MODULE_HEADER_EX**.
[3] This field is filled in by **STANDARD_MODULE_HEADER**.
[4] This field is filled in by **STANDARD_MODULE_PROPERTIES**.
[5] This field is filled in by **NO_MODULE_GLOBALS**.
[6] This field is filled in by **PHP_MODULE_GLOBALS**.
[7] This field only exists when **USING_ZTS** is **TRUE**.
[8] This field only exists when **USING_ZTS** is **FALSE**.
[9] This field is filled in by **STANDARD_MODULE_PROPERTIES_EX**.

Filling in the structure in a practical situation

With all these fields to play with, it can be confusing to know which to use for what purpose. Here is the *zend_module* definition from the "counter" example extension after updating it to its final form.

Example #12 - Counter extension module definition

```
/* {{{ counter_module_entry
*/
zend_module_entry counter_module_entry = {
   STANDARD_MODULE_HEADER,
   "counter",
   counter_functions,
   PHP_MINIT(counter),
   PHP_MSHUTDOWN(counter),
   PHP_RINIT(counter),
   PHP_RSHUTDOWN(counter),
   PHP_MINFO(counter),
   NO_VERSION_YET,
   PHP_MODULE_GLOBALS(counter),
   PHP_GINIT(counter),
   PHP_GSHUTDOWN(counter),
   NULL,
   STANDARD_MODULE_PROPERTIES_EX
};
/* }}} */
```

- **STANDARD_MODULE_HEADER** is used since this module doesn't define any dependencies.
- "counter" is the extension's name, and is used to define the various callback functions

the module passes to Zend. "counter" uses module, globals, and request functions at startup and shutdown times, and provides information to phpinfo(), so all seven callbacks are defined.

- It is assumed that there is a variable of type zend_function_entry * named *counter_functions* earlier in the file that contains the module definition, listing the functions the module exports to userspace.
- **NO_VERSION_YET** is a particularly nice way of telling Zend the module doesn't have a version. It might have been more correct to place *"1.0"* here instead in a real module.
- "counter" uses per-module globals, so **PHP_MODULE_GLOBALS** is used
- This module has no post-deactivate function, so **NULL** is used.
- Since this module *does* use globals, **STANDARD_MODULE_PROPERTIES_EX** is used to finish the structure.

What's changed between 5.2 and 5.3?

Nothing. The only differences in the *zend_module* structure between PHP 5.2 and PHP 5.3 are a few const keywords.

Extension globals

Introduction to globals in a PHP extension

In a language such as C, a "global" variable is a variable that can be accessed from any function without any extra declaration. These traditional globals have a few drawbacks:

- Barring any special options passed to the compiler, a global varaible can be accessed and changed by any piece of code anywhere in the program, whether or not that code should be doing so.
- A typical global variable is not thread safe.
- The names of global variables are as global as the variables themselves.

A PHP extension's globals are more properly called the "extension state", since most modules must remember what they're doing between function calls. The "counter" extension is a perfect example of this need: The basic interface calls for a counter with a persistant value. A programmer new to Zend and PHP might do something like this in *counter.c* to store that value:

Example #13 - The wrong way to store the basic counter interface's value

```
/* ... */
static long basic_counter_value;
```

```
/* ... */

PHP_FUNCTION(counter_get)
{
   RETURN_LONG(basic_counter_value);
}
```

On the surface this appears a viable solution, and indeed in a simple test it would function correctly. However, there are a number of situations in which more than one copy of PHP is running in the same thread, which means more than one instance of the counter module. Suddenly these multiple threads are sharing the same counter value, which is clearly undesireable. Another problem shows itself when considering that another extension might someday happen to have a global with the same name, and due to the rules of C scoping, this has the potential to cause a compile failure, or worse, a runtime error. Something more elaborate is needed, and so exists Zend's support for threadsafe per-module globals.

Declaring module globals

Whether a module uses only a single global or dozens, they must be defined in a structure, and that structure must be declared. There are some macros that assist with doing so in a way that avoids name conflicts between modules: **ZEND_BEGIN_MODULE_GLOBALS()**, **ZEND_END_MODULE_GLOBALS()**, and **ZEND_DECLARE_MODULE_GLOBALS()**. All three take as a parameter the short name of the module, which in the case of the counter module is simply *"counter"*. Here is the global structure declaration from *php_counter.h*:

Example #14 - The counter module's globals

```
ZEND_BEGIN_MODULE_GLOBALS(counter)
   long        basic_counter_value;
ZEND_END_MODULE_GLOBALS(counter)
```

And this is the declaration from *counter.c*:

Example #15 - The counter module's global structure declaration

```
ZEND_DECLARE_MODULE_GLOBALS(counter)
```

Accessing module globals

As discussed above, per-module globals are declared inside a C structure whose name is obscured by Zend macros. As a result, the ideal way to access members of this structure is by the use of further macros. Accordingly, most if not all extensions which have globals have a declaration like this somewhere in their header file:

Example #16 - Accessor macros for per-module globals

```
#ifdef ZTS
#define COUNTER_G(v) TSRMG(counter_globals_id, zend_counter_globals *, v)
#else
#define COUNTER_G(v) (counter_globals.v)
#endif
```

Note

This could have been generalized into a macro of its own by the Zend API, but as of PHP 5.3 (and PHP 6 at the time of this writing), that hasn't happened. The global accessor construct is written into the header by *ext_skel* and thus is generally left alone by extension writers, unless they wish to change the name of the accessor macro.

Note

COUNTER_G was the name given to the macro by *ext_skel*, but it's not necessary for it to have that name and could just as easily be called *FOO* instead.

Any code in the counter extension that accesses a global must thus wrap it in the macro **COUNTER_G**.

Warning

Any function which accesses globals must either be declared by Zend macros, have **TSRMLS_DC** as its last argument, or call the macro **TSRMLS_FETCH** before accessing the globals. See the TSRM documentation for more information.

Life cycle of an extension

Testing an extension

Memory management

Working with variables

Writing functions

PHP is also known as a Glue language, and extending it, can be easily done with those extensions generators. When you use ext_skel and a prototype file to generate the C function stubs, you will notice that all of the exported functions created have a simple prototype such as the following: PHP_FUNCTION(func_name)

Working with classes and objects

Working with resources

Working with INI settings

Working with streams

Note
Information on using streams within the PHP source code can be found in the Streams API for PHP Extension Authors reference.

PDO Driver How-To

The purpose of this How-To is to provide a basic understanding of the steps required to write a database driver that interfaces with the PDO layer. Please note that this is still an evolving API and as such, subject to change. This document was prepared based on version 0.3 of PDO. The learning curve is steep; expect to spend a lot of time on the prerequisites.

Prerequisites

The following is list of prerequisites and assumptions needed for writing a PDO database driver:

- A working target database, examples, demos, etc. working as per vendor specifications;
- A working development environment:
  - Linux: standard development tools, gcc, ld, make, autoconf, automake, etc., versions dependent on distribution;
  - Other Unix: standard development tools supplied by vendor plus the GNU development tool set;
  - Win32: Visual Studio compiler suite;
- A working PHP environment version 5.0.3 or higher with a working PEAR extension version 1.3.5 or higher;
- A working PDO environment (can be installed using 'sudo pecl install PDO'), including the headers which will be needed to access the PDO type definitions and function declarations;
- A good working knowledge of the C programming language;
- A good working knowledge of the way to write a PHP extension; *George Schlossnagle's Advanced PHP Programming* (published by Developer's Library, chapters 21 and 22) is recommended;
- Finally, a familiarity with the Zend API that forms the heart of PHP, in particular paying attention to the memory management aspects.

Preparation and Housekeeping

Source directory layout

The source directory for a typical PDO driver is laid out as follows, where *SKEL* represents

a shortened form of the name of the database that the driver is going to connect to. Even though SKEL is presented here in uppercase (for clarity), the convention is to use lowercase characters.

```
pdo_SKEL/
 config.m4                  # unix build script
 config.w32                 # win32 build script
 CREDITS
 package.xml                # meta information about the package
 pdo_SKEL.c                 # standard PHP extension glue
 php_pdo_SKEL.h
 php_pdo_SKEL_int.h         # driver private header
 SKEL_dbh.c                 # contains the implementation of the PDO driver
interface
 SKEL_stmt.c                # contains the implementation of the PDO statement
interface
 tests/
```

The contents of these files are defined later in this document.

Creating a skeleton

The easiest way to get started is to use the *ext_skel* shell script found in the PHP build tree in the *ext* directory. This will build a skeleton directory containing a lot of the files listed above. It can be build by executing the following command from within the *ext* directory:

```
./ext_skel --extname=pdo_SKEL
```

This will generate a directory called pdo_SKEL containing the skeleton files that you can then modify. This directory should then be moved out of the php extension directory . PDO is a PECL extension and should not be included in the standard extension directory. As long as you have PHP and PDO installed, you should be able to build from any directory.

Standard Includes

Build Specific Headers

The header file config.h is generated by the configure process for the platform for the which the driver is being built. If this header is present, the HAVE_CONFIG_H compiler variable is set. This variable should be tested for and if set, the file config.h should be included in the compilation unit.

PHP Headers

The following standard public php headers should be included in each source module:

- php.h
- php_ini.h

- ext/standard/info.h

PDO Interface Headers

The following standard public PDO header files are also included in each source module:

pdo/php_pdo.h
: This header file contains definitions of the initialization and shutdown functions in the main driver as well as definitions of global PDO variables.

pdo/php_pdo_driver.h
: This header contains the types and API contracts that are used to write a PDO driver. It also contains method signature for calling back into the PDO layer and registering/unregistering your driver with PDO. Most importantly, this header file contains the type definitions for PDO database handles and statements. The two main structures a driver has to deal with, pdo_dbh_t and pdo_stmt_t, are described in more detail in Appendix A and B.

Driver Specific Headers

The typical PDO driver has two header files that are specific to the database implementation. This does not preclude the use of more depending on the implementation. The following two headers are, by convention, standard:

php_pdo_SKEL.h
: This header file is virtually an exact duplicate in functionality and content of the previously defined pdo/php_pdo.h that has been specifically tailored for your database. If your driver requires the use of global variables they should be defined using the ZEND_BEGIN_MODULE_GLOBALS and ZEND_END_MODULE_GLOBALS macros. Macros are then used to access these variables. This macro is usually named PDO_SKEL_G(v) where v is global variable to be accessed. Consult the Zend programmer documentation for more information.

php_pdo_SKEL_int.h
: This header file typically contains type definitions and function declarations specific to the driver implementation. It also should contain the db specific definitions of a pdo_SKEL_handle and pdo_SKEL_stmt structures. These are the names of the private data structures that are then referenced by the driver_data members of the handle and statement structures.

Optional Headers

Depending on the implementation details for a particular driver it may be necessary to include the following header:

```
#include <zend_exceptions.h>
```

Fleshing out your skeleton

Major Structures and Attributes

The major structures, pdo_dbh_t and pdo_stmt_t are defined and explained in Appendix A and B respectively. Database and Statement attributes are defined in Appendix C. Error handling is explained in Appendix D.

pdo_SKEL.c: PHP extension glue

function entries

```
static function_entry pdo_SKEL_functions[] = {
 { NULL, NULL, NULL }
};
```

This structure is used to register functions into the global php function namespace. PDO drivers should try to avoid doing this, so it is recommended that you leave this structure initialized to NULL, as shown in the synopsis above.

Module entry

```
/* {{{ pdo_SKEL_module_entry */
#if ZEND_EXTENSION_API_NO >= 220050617
static zend_module_dep pdo_SKEL_deps[] = {
   ZEND_MOD_REQUIRED("pdo")
   {NULL, NULL, NULL}
};
#endif
/* }}} */

zend_module_entry pdo_SKEL_module_entry = {
#if ZEND_EXTENSION_API_NO >= 220050617
   STANDARD_MODULE_HEADER_EX, NULL,
   pdo_SKEL_deps,
#else
   STANDARD_MODULE_HEADER,
#endif
   "pdo_SKEL",
   pdo_SKEL_functions,
   PHP_MINIT(pdo_SKEL),
   PHP_MSHUTDOWN(pdo_SKEL),
   NULL,
   NULL,
   PHP_MINFO(pdo_SKEL),
   PHP_PDO_<DB>_MODULE_VERSION,
   STANDARD_MODULE_PROPERTIES
};
/* }}} */

#ifdef COMPILE_DL_PDO_<DB>
ZEND_GET_MODULE(pdo_db)
```

```
#endif
```

A structure of type zend_module_entry called pdo_SKEL_module_entry must be declared and should include reference to the pdo_SKEL_functions table defined previously.

Standard PHP Module Extension Functions

PHP_MINIT_FUNCTION

```
/* {{{ PHP_MINIT_FUNCTION */
PHP_MINIT_FUNCTION(pdo_SKEL)
{
   return php_pdo_register_driver(&pdo_SKEL_driver);
}
/* }}} */
```

This standard PHP extension function should be used to register your driver with the PDO layer. This is done by calling the **php_pdo_register_driver()** function passing a pointer to a structure of type pdo_driver_t typically named *pdo_SKEL_driver*. A pdo_driver_t contains a header that is generated using the *PDO_DRIVER_HEADER(SKEL)* macro and **pdo_SKEL_handle_factory()** function pointer. The actual function is described during the discussion of the *SKEL_dbh.c* unit.

PHP_MSHUTDOWN_FUNCTION

```
/* {{{ PHP_MSHUTDOWN_FUNCTION */
PHP_MSHUTDOWN_FUNCTION(pdo_SKEL)
{
   php_pdo_unregister_driver(&pdo_SKEL_driver);
   return SUCCESS;
}
/* }}} */
```

This standard PHP extension function is used to unregister your driver from the PDO layer. This is done by calling the **php_pdo_unregister_driver()** function, passing the same *pdo_SKEL_driver* structure that was passed in the init function above.

PHP_MINFO_FUNCTION

This is again a standard PHP extension function. Its purpose is to display information regarding the module when the phpinfo() is called from a script. The convention is to display the version of the module and also what version of the db you are dependent on, along with any other configuration style information that might be relevant.

SKEL_driver.c: Driver implementation

This unit implements all of the database handling methods that support the PDO database handle object. It also contains the error fetching routines. All of these functions will typically need to access the global variable pool. Therefore, it is necessary to use the Zend macro

TSRMLS_DC macro at the end of each of these statements. Consult the Zend programmer documentation for more information on this macro.

pdo_SKEL_error

```
static int pdo_SKEL_error(pdo_dbh_t *dbh,
 pdo_stmt_t *stmt, const char *file, int line TSRMLS_DC)
```

The purpose of this function is to be used as a generic error handling function within the driver. It is called by the driver when an error occurs within the driver. If an error occurs that is not related to SQLSTATE, the driver should set either *dbh->error_code* or *stmt->error_code* to an SQLSTATE that most closely matches the error or the generic SQLSTATE error "HY000". The file pdo_sqlstate.c in the PDO source contains a table of commonly used SQLSTATE codes that the PDO code explicitly recognizes. This setting of the error code should be done prior to calling this function.; This function should set the global *pdo_err* variable to the error found in either the dbh or the stmt (if the variable stmt is not NULL).

dbh
: Pointer to the database handle initialized by the handle factory

stmt
: Pointer to the current statement or NULL. If NULL, the error is derived by error code found in the dbh.

file
: The source file where the error occurred or NULL if not available.

line
: The line number within the source file if available.

If the dbh member methods is NULL (which implies that the error is being raised from within the PDO constructor), this function should call the zend_throw_exception_ex() function otherwise it should return the error code. This function is usually called using a helper macro that customizes the calling sequence for either database handling errors or statement handling errors.

Example #17 - Example macros for invoking pdo_SKEL_error

```
#define pdo_SKEL_drv_error(what) \
   pdo_SKEL_error(dbh, NULL, what, __FILE__, __LINE__ TSRMLS_CC)
#define pdo_SKEL_drv_error(what) \
   pdo_SKEL_error(dbh, NULL, what, __FILE__, __LINE__ TSRMLS_CC)
```

For more info on error handling, see Error handling.

Note

Despite being documented here, the PDO driver interface does not specify that this

function be present; it is merely a convenient way to handle errors, and it just happens to be equally convenient for the majority of database client library APIs to structure your driver implementation in this way.

pdo_SKEL_fetch_error_func

```
static int pdo_SKEL_fetch_error_func(pdo_dbh_t *dbh, pdo_stmt_t *stmt,
   zval *info TSRMLS_DC)
```

The purpose of this function is to obtain additional information about the last error that was triggered. This includes the driver specific error code and a human readable string. It may also include additional information if appropriate. This function is called as a result of the PHP script calling the PDO::errorInfo() method.

dbh
: Pointer to the database handle initialized by the handle factory

stmt
: Pointer to the most current statement or NULL. If NULL, the error translated is derived by error code found in the dbh.

info
: A hash table containing error codes and messages.

The error_func should return two pieces of information as successive array elements. The first item is expected to be a numeric error code, the second item is a descriptive string. The best way to set this item is by using add_next_index. Note that the type of the first argument need not be long; use whichever type most closely matches the error code returned by the underlying database API.

```
/* now add the error information. */
/* These need to be added in a specific order */
add_next_index_long(info, error_code);   /* driver specific error code */
add_next_index_string(info, message, 0); /* readable error message */
```

This function should return 1 if information is available, 0 if the driver does not have additional info.

SKEL_handle_closer

```
static int SKEL_handle_closer(pdo_dbh_t *dbh TSRMLS_DC)
```

This function will be called by PDO to close an open database.

dbh
: Pointer to the database handle initialized by the handle factory

This should do whatever database specific activity that needs to be accomplished to close the open database. PDO ignores the return value from this function.

SKEL_handle_preparer

```
static int SKEL_handle_preparer(pdo_dbh_t *dbh, const char *sql,
long sql_len, pdo_stmt_t *stmt, zval *driver_options TSRMLS_DC)
```

This function will be called by PDO in response to PDO::query() and PDO::prepare() calls from the PHP script. The purpose of the function is to prepare raw SQL for execution, storing whatever state is appropriate into the *stmt* that is passed in.

dbh
: Pointer to the database handle initialized by the handle factory

sql
: Pointer to a character string containing the SQL statement to be prepared.

sql_len
: The length of the SQL statement.

Stmt
: Pointer to the returned statement or NULL if an error occurs.

driver_options
: Any driver specific/defined options.

This function is essentially the constructor for a stmt object. This function is responsible for processing statement options, and setting driver-specific option fields in the pdo_stmt_t structure.

PDO does not process any statement options on the driver's behalf before calling the preparer function. It is your responsibility to process them before you return, raising an error for any unknown options that are passed.

One very important responsibility of this function is the processing of SQL statement parameters. At the time of this call, PDO does not know if your driver supports binding parameters into prepared statements, nor does it know if it supports named or positional parameter naming conventions.

Your driver is responsible for setting *stmt->supports_placeholders* as appropriate for the underlying database. This may involve some run-time determination on the part of your driver, if this setting depends on the version of the database server to which it is connected. If your driver doesn't directly support both named and positional parameter conventions, you should use the **pdo_parse_params()** API to have PDO rewrite the query to take advantage of the support provided by your database.

Example #18 - Using pdo_parse_params

```
int ret;
   char *nsql = NULL;
   int nsql_len = 0;

   /* before we prepare, we need to peek at the query; if it uses named
parameters,
```

```
     * we want PDO to rewrite them for us */
    stmt->supports_placeholders = PDO_PLACEHOLDER_POSITIONAL;
    ret = pdo_parse_params(stmt, (char*)sql, sql_len, &nsql, &nsql_len
TSRMLS_CC);

    if (ret == 1) {
        /* query was re-written */
        sql = nsql;
    } else if (ret == -1) {
        /* couldn't grok it */
        strcpy(dbh->error_code, stmt->error_code);
        return 0;
    }

    /* now proceed to prepare the query in "sql" */
```

Possible values for *supports_placeholders* are: **PDO_PLACEHOLDER_NAMED**, **PDO_PLACEHOLDER_POSITIONAL** and **PDO_PLACEHOLDER_NONE**. If the driver doesn't support prepare statements at all, then this function should simply allocate any state that it might need, and then return:

Example #19 - Implementing preparer for drivers that don't support native prepared statements

```
static int SKEL_handle_preparer(pdo_dbh_t *dbh, const char *sql,
    long sql_len, pdo_stmt_t *stmt, zval *driver_options TSRMLS_DC)
{
    pdo_SKEL_db_handle *H = (pdo_SKEL_db_handle *)dbh->driver_data;
    pdo_SKEL_stmt *S = ecalloc(1, sizeof(pdo_SKEL_stmt));

    S->H = H;
    stmt->driver_data = S;
    stmt->methods = &SKEL_stmt_methods;
    stmt->supports_placeholders = PDO_PLACEHOLDER_NONE;

    return 1;
}
```

This function returns 1 on success or 0 on failure.

SKEL_handle_doer

```
static long SKEL_handle_doer(pdo_dbh_t *dbh, const char *sql, long sql_len
TSRMLS_DC)
```

This function will be called by PDO to execute a raw SQL statement. No pdo_stmt_t is created.

dbh
: Pointer to the database handle initialized by the handle factory

sql

Pointer to a character string containing the SQL statement to be prepared.

sql_len
The length of the SQL statement.

This function returns 1 on success or 0 on failure.

SKEL_handle_quoter

```
static int SKEL_handle_quoter(pdo_dbh_t *dbh, const char *unquoted,
 int unquoted_len, char **quoted, int quoted_len, enum pdo_param_type param_type
TSRMLS_DC)
```

This function will be called by PDO to turn an unquoted string into a quoted string for use in a query.

dbh
Pointer to the database handle initialized by the handle factory

unquoted
Pointer to a character string containing the string to be quoted.

unquoted_len
The length of the string to be quoted.

quoted
Pointer to the address where a pointer to the newly quoted string will be returned.

quoted_len
The length of the new string.

param_type
A driver specific hint for driver that have alternate quoting styles

This function is called in response to a call to PDO::quote() or when the driver has set *supports_placeholder* to **PDO_PLACEHOLDER_NONE**. The purpose is to quote a parameter when building SQL statements.

If your driver does not support native prepared statements, implementation of this function is required.

This function returns 1 if the quoting process reformatted the string, and 0 if it was not necessary to change the string. The original string will be used unchanged with a 0 return.

SKEL_handle_begin

```
static int SKEL_handle_begin(pdo_dbh_t *dbh TSRMLS_DC)
```

This function will be called by PDO to begin a database transaction.

dbh

Pointer to the database handle initialized by the handle factory

This should do whatever database specific activity that needs to be accomplished to begin a transaction. This function returns 1 for success or 0 if an error occurred.

SKEL_handle_commit

```
static int SKEL_handle_commit(pdo_dbh_t *dbh TSRMLS_DC)
```

This function will be called by PDO to end a database transaction.

dbh
Pointer to the database handle initialized by the handle factory

This should do whatever database specific activity that needs to be accomplished to commit a transaction. This function returns 1 for success or 0 if an error occurred.

SKEL_handle_rollback

```
static int SKEL_handle_rollback( pdo_dbh_t *dbh TSRMLS_DC)
```

This function will be called by PDO to rollback a database transaction.

dbh
Pointer to the database handle initialized by the handle factory

This should do whatever database specific activity that needs to be accomplished to rollback a transaction. This function returns 1 for success or 0 if an error occurred.

SKEL_handle_get_attribute

```
static int SKEL_handle_get_attribute(pdo_dbh_t *dbh, long attr, zval
*return_value TSRMLS_DC)
```

This function will be called by PDO to retrieve a database attribute.

dbh
Pointer to the database handle initialized by the handle factory

attr
long value of one of the PDO_ATTR_xxxx types. See Database and Statement Attributes Table for valid attributes.

return_value
The returned value for the attribute.

It is up to the driver to decide which attributes will be supported for a particular implementation. It is not necessary for a driver to supply this function. PDO driver handles the PDO_ATTR_PERSISTENT, PDO_ATTR_CASE, PDO_ATTR_ORACLE_NULLS, and PDO_ATTR_ERRMODE attributes directly.

This function returns 1 on success or 0 on failure.

SKEL_handle_set_attribute

```
static int SKEL_handle_set_attribute(pdo_dbh_t *dbh, long attr, zval *val
TSRMLS_DC)
```

This function will be called by PDO to set a database attribute, usually in response to a script calling PDO::setAttribute().

dbh
: Pointer to the database handle initialized by the handle factory

attr
: long value of one of the PDO_ATTR_xxxx types. See Database and Statement Attributes Table for valid attributes.

val
: The new value for the attribute.

It is up to the driver to decide which attributes will be supported for a particular implementation. It is not necessary for a driver to provide this function if it does not need to support additional attributes. The PDO driver handles the PDO_ATTR_CASE, PDO_ATTR_ORACLE_NULLS, and PDO_ATTR_ERRMODE attributes directly.

This function returns 1 on success or 0 on failure.

SKEL_handle_last_id

```
static char * SKEL_handle_last_id(pdo_dbh_t *dbh, const char *name, unsigned int
len TSRMLS_DC)
```

This function will be called by PDO to retrieve the ID of the last inserted row.

dbh
: Pointer to the database handle initialized by the handle factory

name
: string representing a table or sequence name.

len
: the length of the *name* parameter.

This function returns a character string containing the id of the last inserted row on success or NULL on failure. This is an optional function.

SKEL_check_liveness

```
static int SKEL_check_liveness(pdo_dbh_t *dbh TSRMLS_DC)
```

This function will be called by PDO to test whether or not a persistent connection to a database is alive and ready for use.

dbh
: Pointer to the database handle initialized by the handle factory

This function returns 1 if the database connection is alive and ready for use, otherwise it should return 0 to indicate failure or lack of support.

Note
This is an optional function.

SKEL_get_driver_methods

```
static function_entry *SKEL_get_driver_methods(pdo_dbh_t *dbh, int kind
TSRMLS_DC)
```

This function will be called by PDO in response to a call to any method that is not a part of either the PDO or PDOStatement classes. It's purpose is to allow the driver to provide additional driver specific methods to those classes.

dbh
: Pointer to the database handle initialized by the handle factory

kind
: One of the following:
  PDO_DBH_DRIVER_METHOD_KIND_DBH
  : Set when the method call was attempted on an instance of the PDO class. The driver should return a pointer a function_entry table for any methods it wants to add to that class, or NULL if there are none.

  PDO_DBH_DRIVER_METHOD_KIND_STMT
  : Set when the method call was attempted on an instance of the PDOStatement class. The driver should return a pointer to a function_entry table for any methods it wants to add to that class, or NULL if there are none.

This function returns a pointer to the function_entry table requested, or NULL there are no driver specific methods.

SKEL_handle_factory

```
static int SKEL_handle_factory(pdo_dbh_t *dbh, zval *driver_options TSRMLS_DC)
```

This function will be called by PDO to create a database handle. For most databases this involves establishing a connection to the database. In some cases, a persistent connection may be requested, in other cases connection pooling may be requested. All of these are

database/driver dependent.

dbh
: Pointer to the database handle initialized by the handle factory

driver_options
: An array of driver options, keyed by integer option number. See Database and Statement Attributes Table for a list of possible attributes.

This function should fill in the passed database handle structure with its driver specific information on success and return 1, otherwise it should return 0 to indicate failure.

PDO processes the AUTOCOMMIT and PERSISTENT driver options before calling the handle_factory. It is the handle factory's responsibility to process other options.

Driver method table

A static structure of type pdo_dbh_methods named SKEL_methods must be declared and initialized to the function pointers for each defined function. If a function is not supported or not implemented the value for that function pointer should be set to NULL.

pdo_SKEL_driver

A structure of type pdo_driver_t named pdo_SKEL_driver should be declared. The PDO_DRIVER_HEADER(SKEL) macro should be used to declare the header and the function pointer to the handle factory function should set.

SKEL_statement.c: Statement implementation

This unit implements all of the database statement handling methods that support the PDO statement object.

SKEL_stmt_dtor

```
static int SKEL_stmt_dtor(pdo_stmt_t *stmt TSRMLS_DC)
```

This function will be called by PDO to destroy a previously constructed statement object.

stmt
: Pointer to the statement structure initialized by SKEL_handle_preparer.

This should do whatever is necessary to free up any driver specific storage allocated for the statement. The return value from this function is ignored.

SKEL_stmt_execute

```
static int SKEL_stmt_execute(pdo_stmt_t *stmt TSRMLS_DC)
```

This function will be called by PDO to execute the prepared SQL statement in the passed statement object.

stmt
: Pointer to the statement structure initialized by SKEL_handle_preparer.

This function returns 1 for success or 0 in the event of failure.

SKEL_stmt_fetch

```
static int SKEL_stmt_fetch(pdo_stmt_t *stmt, enum pdo_fetch_orientation ori,
  long offset TSRMLS_DC)
```

This function will be called by PDO to fetch a row from a previously executed statement object.

stmt
: Pointer to the statement structure initialized by SKEL_handle_preparer.

ori
: One of PDO_FETCH_ORI_xxx which will determine which row will be fetched.

offset
: If ori is set to PDO_FETCH_ORI_ABS or PDO_FETCH_ORI_REL, offset represents the row desired or the row relative to the current position, respectively. Otherwise, this value is ignored.

The results of this fetch are driver dependent and the data is usually stored in the driver_data member of the pdo_stmt_t object. The ori and offset parameters are only meaningful if the statement represents a scrollable cursor. This function returns 1 for success or 0 in the event of failure.

SKEL_stmt_param_hook

```
static int SKEL_stmt_param_hook(pdo_stmt_t *stmt,
 struct pdo_bound_param_data *param, enum pdo_param_event event_type TSRMLS_DC)
```

This function will be called by PDO for handling of both bound parameters and bound columns.

stmt
: Pointer to the statement structure initialized by SKEL_handle_preparer.

param
: The structure describing either a statement parameter or a bound column.

event_type
: The type of event to occur for this parameter, one of the following:
  PDO_PARAM_EVT_ALLOC
  : Called when PDO allocates the binding. Occurs as part of PDOStatement::bindParam(), PDOStatement::bindValue() or as part of an implicit

bind when calling PDOStatement::execute(). This is your opportunity to take some action at this point; drivers that implement native prepared statements will typically want to query the parameter information, reconcile the type with that requested by the script, allocate an appropriately sized buffer and then bind the parameter to that buffer. You should not rely on the type or value of the zval at *param->parameter* at this point in time.

PDO_PARAM_EVT_FREE
: Called once per parameter as part of cleanup. You should release any resources associated with that parameter now.

PDO_PARAM_EXEC_PRE
: Called once for each parameter immediately before calling SKEL_stmt_execute; take this opportunity to make any final adjustments ready for execution. In particular, you should note that variables bound via PDOStatement::bindParam() are only legal to touch now, and not any sooner.

PDO_PARAM_EXEC_POST
: Called once for each parameter immediately after calling SKEL_stmt_execute; take this opportunity to make any post-execution actions that might be required by your driver.

PDO_PARAM_FETCH_PRE
: Called once for each parameter immediately prior to calling SKEL_stmt_fetch.

PDO_PARAM_FETCH_POST
: Called once for each parameter immediately after calling SKEL_stmt_fetch.

This hook will be called for each bound parameter and bound column in the statement. For ALLOC and FREE events, a single call will be made for each parameter or column. The param structure contains a driver_data field that the driver can use to store implementation specific information about each of the parameters.

For all other events, PDO may call you multiple times as the script issues PDOStatement::execute() and PDOStatement::fetch() calls.

If this is a bound parameter, the is_param flag in the param structure is set, otherwise the param structure refers to a bound column.

This function returns 1 for success or 0 in the event of failure.

SKEL_stmt_describe_col

```
static int SKEL_stmt_describe_col(pdo_stmt_t *stmt, int colno TSRMLS_DC)
```

This function will be called by PDO to query information about a particular column.

stmt
: Pointer to the statement structure initialized by SKEL_handle_preparer.

colno
: The column number to be queried.

The driver should populate the pdo_stmt_t member columns(colno) with the appropriate information. This function returns 1 for success or 0 in the event of failure.

SKEL_stmt_get_col_data

```
static int SKEL_stmt_get_col_data(pdo_stmt_t *stmt, int colno,
 char **ptr, unsigned long *len, int *caller_frees TSRMLS_DC)
```

This function will be called by PDO to retrieve data from the specified column.

stmt
: Pointer to the statement structure initialized by SKEL_handle_preparer.

colno
: The column number to be queried.

ptr
: Pointer to the retrieved data.

len
: The length of the data pointed to by ptr.

caller_frees
: If set, ptr should point to emalloc'd memory and the main PDO driver will free it as soon as it is done with it. Otherwise, it will be the responsibility of the driver to free any allocated memory as a result of this call.

The driver should return the resultant data and length of that data in the ptr and len variables respectively. It should be noted that the main PDO driver expects the driver to manage the lifetime of the data. This function returns 1 for success or 0 in the event of failure.

SKEL_stmt_set_attr

```
static int SKEL_stmt_set_attr(pdo_stmt_t *stmt, long attr, zval *val TSRMLS_DC)
```

This function will be called by PDO to allow the setting of driver specific attributes for a statement object.

stmt
: Pointer to the statement structure initialized by SKEL_handle_preparer.

attr
: long value of one of the PDO_ATTR_xxxx types. See Database and Statement Attributes Table for valid attributes.

val

The new value for the attribute.

This function is driver dependent and allows the driver the capability to set database specific attributes for a statement. This function returns 1 for success or 0 in the event of failure. This is an optional function. If the driver does not support additional settable attributes, it can be NULLed in the method table. The PDO driver does not handle any settable attributes on the database driver's behalf.

SKEL_stmt_get_attr

```
static int SKEL_stmt_get_attr(pdo_stmt_t *stmt, long attr, zval
  *return_value TSRMLS_DC)
```

This function will be called by PDO to allow the retrieval of driver specific attributes for a statement object.

stmt
: Pointer to the statement structure initialized by SKEL_handle_preparer.

attr
: long value of one of the PDO_ATTR_xxxx types. See Database and Statement Attributes Table for valid attributes.

return_value
: The returned value for the attribute.

This function is driver dependent and allows the driver the capability to retrieve a previously set database specific attribute for a statement. This function returns 1 for success or 0 in the event of failure. This is an optional function. If the driver does not support additional gettable attributes, it can be NULLed in the method table. The PDO driver does not handle any settable attributes on the database driver's behalf.

SKEL_stmt_get_col_meta

```
static int SKEL_stmt_get_col_meta(pdo_stmt_t *stmt, int colno,
  zval *return_value TSRMLS_DC)
```

Warning
This function is not well defined and is subject to change.

This function will be called by PDO to retrieve meta data from the specified column.

stmt
: Pointer to the statement structure initialized by SKEL_handle_preparer.

colno
: The column number for which data is to be retrieved.

return_value
: Holds the returned meta data.

The driver author should consult the documentation for this function that can be found in the php_pdo_driver.h header as this will be the most current. This function returns 1 for success or 0 in the event of failure. The database driver does not need to provide this function.

Statement handling method table

A static structure of type pdo_stmt_methods named SKEL_stmt_methods should be declared and initialized to the function pointers for each defined function. If a function is not supported or not implemented the value for that function pointer should be set to NULL.

Building

The build process is designed to work with PEAR (see for more information about PEAR). There are two files that are used to assist in configuring your package for building. The first is config.m4 which is the *autoconf* configuration file for all platforms except Win32. The second is config.w32 which is a build configuration file for use on Win32. Skeleton files for these are built for you when you first set up your project. You then need to customize them to fit the needs of your project. Once you've customized your config files, you can build your driver using the following sequence of commands:

Before first build:

```
$ sudo pecl install PDO
```

For each build:

```
$ cd pdo_SKEL
$ phpize
$ ./configure
$ make
$ sudo make install
```

The process can then be repeated as necessary during the development process.

Testing

PDO has a set of "core" tests that all drivers should pass before being released. They're designed to run from the PHP source distribution, so running the tests for your driver requires moving things around a bit. The suggested procedure is to obtain the latest PHP 5.1 snapshot and perform the following step:

```
$ cp -r pdo_SKEL /path/to/php-5.1/ext
```

This will allow the test harness to run your tests. The next thing you need to do is create a test that will redirect into the PDO common core tests. The convention is to name this file

common.phpt; it should be placed in the tests subdirectory that was created by *ext_skel* when you created your extension skeleton. The content of this file should look something like the following:

```
--TEST--
SKEL
--SKIPIF--
<?php # vim:ft=php
if (!extension_loaded('pdo_SKEL')) print 'skip'; ?>
--REDIRECTTEST--
if (false !== getenv('PDO_SKEL_TEST_DSN')) {
# user set them from their shell
  $config['ENV']['PDOTEST_DSN'] = getenv('PDO_SKEL_TEST_DSN');
  $config['ENV']['PDOTEST_USER'] = getenv('PDO_SKEL_TEST_USER');
  $config['ENV']['PDOTEST_PASS'] = getenv('PDO_SKEL_TEST_PASS');
  if (false !== getenv('PDO_SKEL_TEST_ATTR')) {
     $config['ENV']['PDOTEST_ATTR'] = getenv('PDO_SKEL_TEST_ATTR');
  }
  return $config;
}
return array(
  'ENV' => array(
          'PDOTEST_DSN' => 'SKEL:dsn',
          'PDOTEST_USER' => 'username',
          'PDOTEST_PASS' => 'password'
      ),
  'TESTS' => 'ext/pdo/tests'
  );
```

This will cause the common core tests to be run, passing the values of *PDOTEST_DSN*, *PDOTEST_USER* and *PDOTEST_PASS* to the PDO constructor as the *`dsn`*, *`username`* and *`password`* parameters. It will first check the environment, so that appropriate values can be passed in when the test harness is run, rather than hard-coding the database credentials into the test file.

The test harness can be invoked as follows:

```
$ cd /path/to/php-5.1
$ make TESTS=ext/pdo_SKEL/tests PDO_SKEL_TEST_DSN="skel:dsn" \
PDO_SKEL_TEST_USER=user PDO_SKEL_TEST_PASS=pass test
```

Packaging and distribution

Creating a package

PDO drivers are released via PECL; all the usual rules for PECL extensions apply. Packaging is accomplished by creating a valid *package.xml* file and then running:

```
$ pecl package
```

This will create a tarball named *PDO_SKEL-X.Y.Z.tgz*.

Before releasing the package, you should test that it builds correctly; if you've made a mistake in your *config.m4* or *package.xml* files, the package may not function correctly.

You can test the build, without installing anything, using the following invocation:

```
$ pecl build package.xml
```

Once this is proven to work, you can test installation:

```
$ pecl package
$ sudo pecl install PDO_SKEL-X.Y.X.tgz
```

Full details about *package.xml* can be found in the PEAR Programmer's documentation ( ).

Releasing the package

A PDO driver is released via the PHP Extension Community Library (PECL). Information about PECL can be found at » http://pecl.php.net/index.php.

pdo_dbh_t definition

All fields should be treated as read-only by the driver, unless explicitly stated otherwise.

pdo_dbh_t

```
/* represents a connection to a database */
struct _pdo_dbh_t {
   /* driver specific methods */
   struct pdo_dbh_methods *methods;     [1]
   /* driver specific data */
   void *driver_data;                   [2]

   /* credentials */
   char *username, *password;           [3]

   /* if true, then data stored and pointed at by this handle must all be
    * persistently allocated */
   unsigned is_persistent:1;            [4]

   /* if true, driver should act as though a COMMIT were executed between
    * each executed statement; otherwise, COMMIT must be carried out manually
    * */
   unsigned auto_commit:1;              [5]

   /* if true, the driver requires that memory be allocated explicitly for
    * the columns that are returned */
   unsigned alloc_own_columns:1;        [6]

   /* if true, commit or rollBack is allowed to be called */
   unsigned in_txn:1;

   /* max length a single character can become after correct quoting */
   unsigned max_escaped_char_length:3;    [7]

   /* data source string used to open this handle */
   const char *data_source;             [8]
```

```
    unsigned long data_source_len;

    /* the global error code. */
    pdo_error_type error_code;          [9]

    enum pdo_case_conversion native_case [10], desired_case;
};
```

[1] The driver *must* set this during **SKEL_handle_factory()**.
[2] This item is for use by the driver; the intended usage is to store a pointer (during **SKEL_handle_factory()**) to whatever instance data is required to maintain a connection to the database.
[3] The username and password that were passed into the PDO constructor. The driver should use these values when it initiates a connection to the database.
[4] If this is set to 1, then any data that is referenced by the dbh, including whatever structure your driver allocates, *MUST* be allocated persistently. This is easy to achieve; rather than using the usual **emalloc()** simply use **pemalloc()** and pass the value of this flag as the last parameter. Failure to use the appropriate kind of memory can lead to serious memory faults, resulting (in the best case) a hard crash, and in the worst case, an exploitable memory problem. If, for whatever reason, your driver is not suitable to run persistently, you *MUST* check this flag in your **SKEL_handle_factory()** and raise an appropriate error.
[5] You should check this value in your **SKEL_handle_doer()** and **SKEL_stmt_execute()** functions; if it evaluates to true, you must attempt to commit the query now. Most database implementations offer an auto-commit mode that handles this automatically.
[6] If your database client library API operates by fetching data into a caller-supplied buffer, you should set this flag to 1 during your **SKEL_handle_factory()**. When set, PDO will call your **SKEL_stmt_describer()** earlier than it would otherwise. This early call allows you to determine those buffer sizes and issue appropriate calls to the database client library. If your database client library API simply returns pointers to its own internal buffers for you to copy after each fetch call, you should leave this value set to 0.
[7] If your driver doesn't support native prepared statements (`supports_placeholders` is set to **PDO_PLACEHOLDER_NONE**), you must set this value to the maximum length that can be taken up by a single character when it is quoted by your **SKEL_handle_quoter()** function. This value is used to calculate the amount of buffer space required when PDO executes the statement.
[8] This holds the value of the DSN that was passed into the PDO constructor. If your driver implementation needed to modify the DSN for whatever reason, it should update this member during **SKEL_handle_factory()**. Modifying this member should be avoided. If you do change it, you must ensure that `data_source_len` is also correct.
[9] Whenever an error occurs during a call to one of your driver methods, you should set this member to the SQLSTATE code that best describes the error and return an error. In this HOW-TO, the suggested practice is to call **SKEL_handle_error()** when an error is detected, and have it set the error code.
[10] Your driver should set this during **SKEL_handle_factory()**; the value should reflect how the database returns the names of the columns in result sets. If the name matches the case that was used in the query, set it to **PDO_CASE_NATURAL** (this is actually the default). If the column names are always returned in upper case, set it to **PDO_CASE_UPPER**. If the column names are always returned in lower case, set it to **PDO_CASE_LOWER**. The value you set is used to determine if PDO should perform case folding when the user sets the **PDO_ATTR_CASE** attribute.

pdo_stmt_t definition

All fields should be treated as read-only unless explicitly stated otherwise.

pdo_stmt_t

```
/* represents a prepared statement */
struct _pdo_stmt_t {
   /* driver specifics */
   struct pdo_stmt_methods *methods;    [1]
   void *driver_data;                   [2]

   /* if true, we've already successfully executed this statement at least
    * once */
   unsigned executed:1;                 [3]
   /* if true, the statement supports placeholders and can implement
    * bindParam() for its prepared statements, if false, PDO should
    * emulate prepare and bind on its behalf */
   unsigned supports_placeholders:2;    [4]

   /* the number of columns in the result set; not valid until after
    * the statement has been executed at least once.  In some cases, might
    * not be valid until fetch (at the driver level) has been called at least
once.
    * */
   int column_count;                    [5]
   struct pdo_column_data *columns;     [6]

   /* points at the dbh that this statement was prepared on */
   pdo_dbh_t *dbh;

   /* keep track of bound input parameters.  Some drivers support
    * input/output parameters, but you can't rely on that working */
   HashTable *bound_params;
   /* When rewriting from named to positional, this maps positions to names */
   HashTable *bound_param_map;
   /* keep track of PHP variables bound to named (or positional) columns
    * in the result set */
   HashTable *bound_columns;

   /* not always meaningful */
   long row_count;

   /* used to hold the statement's current query */
   char *query_string;
   int query_stringlen;

   /* the copy of the query with expanded binds ONLY for emulated-prepare
drivers */
   char *active_query_string;
   int active_query_stringlen;

   /* the cursor specific error code. */
   pdo_error_type error_code;

   /* used by the query parser for driver specific
    * parameter naming (see pgsql driver for example) */
```

```
    const char *named_rewrite_template;
};
```

[1] The driver *must* set this during **SKEL_handle_preparer()**.
[2] This item is for use by the driver; the intended usage is to store a pointer (during **SKEL_handle_factory()**) to whatever instance data is required to maintain a connection to the database.
[3] This is set by PDO after the statement has been executed for the first time. Your driver can inspect this value to determine if it can skip one-time actions as an optimization.
[4] Discussed in more detail in SKEL_handle_preparer.
[5] Your driver is responsible for setting this field to the number of columns available in a result set. This is usually set during **SKEL_stmt_execute()** but with some database implementations, the column count may not be available until **SKEL_stmt_fetch()** has been called at least once. Drivers that implement **SKEL_stmt_next_rowset()** should update the column count when a new rowset is available.
[6] PDO will allocate this field based on the value that you set for the column count. You are responsible for populating each column during **SKEL_stmt_describe()**. You must set the *`precision`*, *`maxlen`*, *`name`*, *`namelen`* and *`param_type`* members for each column. The *`name`* is expected to be allocated using **emalloc()**; PDO will call **efree()** at the appropriate time.

Constants

Database and Statement Attributes Table

Attribute	Valid value(s)
PDO_ATTR_AUTOCOMMIT	BOOL TRUE if autocommit is set, FALSE otherwise. dbh->auto_commit contains value. Processed by PDO directly.
PDO_ATTR_PREFETCH	LONG Value of the prefetch size in drivers that support it.

PDO_ATTR_TIMEOUT	LONG How long to wait for a db operation before timing out.
PDO_ATTR_ERRMODE	LONG Processed and handled by PDO
PDO_ATTR_SERVER_VERSION	STRING The "human-readable" string representing the Server/Version this driver is currently connected to.
PDO_ATTR_CLIENT_VERSION	STRING The "human-readable" string representing the Client/Version this driver supports.
PDO_ATTR_SERVER_INFO	STRING The "human-readable" description of the Server.
PDO_ATTR_CONNECTION_STATUS	LONG Values not yet defined
PDO_ATTR_CASE	LONG Processed and handled by PDO.
PDO_ATTR_CURSOR_NAME	STRING String representing the name for a database cursor for use in "where current in <name>" SQL statements.

PDO_ATTR_CURSOR	LONG PDO_CURSOR_FWDONLY Forward only cursor PDO_CURSOR_SCROLL Scrollable cursor

The values for the attributes above are all defined in terms of the Zend API. The Zend API contains macros that can be used to convert a *zval to a value. These macros are defined in the Zend header file, zend_API.h in the Zend directory of your PHP build directory. Some of these attributes can be used with the statement attribute handlers such as the PDO_ATTR_CURSOR and PDO_ATTR_CURSOR_NAME. See the statement attribute handling functions for more information.

Error handling

Error handling is implemented using a hand-shaking protocol between PDO and the database driver code. The database driver code signals PDO that an error has occurred via a failure (*0*) return from any of the interface functions. If a zero is returned, set the field `error_code` in the control block appropriate to the context (either the pdo_dbh_t or pdo_stmt_t block). In practice, it is probably a good idea to set the field in both blocks to the same value to ensure the correct one is getting used.

The error_mode field is a six-byte field containing a 5 character ASCIIZ SQLSTATE identifier code. This code drives the error message process. The SQLSTATE code is used to look up an error message in the internal PDO error message table (see pdo_sqlstate.c for a list of error codes and their messages). If the code is not known to PDO, a default "Unknown Message" value will be used.

In addition to the SQLSTATE code and error message, PDO will call the driver-specific fetch_err() routine to obtain supplemental data for the particular error condition. This routine is passed an array into which the driver may place additional information. This array has slot positions assigned to particular types of supplemental info:

- A native error code. This will frequently be an error code obtained from the database API.
- A descriptive string. This string can contain any additional information related to the failure. Database drivers typically include information such as an error message, code location of the failure, and any additional descriptive information the driver developer feels worthy of inclusion. It is generally a good idea to include all diagnostic information obtainable from the database interface at the time of the failure. For driver-detected errors (such as memory allocation problems), the driver developer can define whatever error information that seems appropriate.

Extension FAQs

Zend Engine 2 API reference

Zend Engine 1

Zend Engine 1 is the internal engine used by PHP for the entire version 4 release line. It is no longer considered active, but PHP 4 is still in widespread use, so the old ZE1 documentation is preserved here exactly as it was.

Old introduction

If you are about to begin developing PHP or Zend extensions, you need to prepare yourself for the programming environment provided by the various APIs. This part of the documentation tries to introduce the APIs provided by the different PHP and Zend Engine versions available. Since most of the information available here is somewhat outdated, you'll want to read various files found in the PHP source, files such as *README.SELF-CONTAINED-EXTENSIONS* and *README.EXT_SKEL* in addition to the manual.

Streams API for PHP Extension Authors

Note
The functions in this chapter are for use in the PHP source code and are not PHP functions. Information on userland stream functions can be found in the Stream Reference.

Overview

The PHP Streams API introduces a unified approach to the handling of files and sockets in PHP extension. Using a single API with standard functions for common operations, the streams API allows your extension to access files, sockets, URLs, memory and script-defined objects. Streams is a run-time extensible API that allows dynamically loaded modules (and scripts!) to register new streams.

The aim of the Streams API is to make it comfortable for developers to open files, URLs and other streamable data sources with a unified API that is easy to understand. The API is more or less based on the ANSI C stdio family of functions (with identical semantics for most of the main functions), so C programmers will have a feeling of familiarity with streams.

The streams API operates on a couple of different levels: at the base level, the API defines php_stream objects to represent streamable data sources. On a slightly higher level, the API defines php_stream_wrapper objects which "wrap" around the lower level API to provide support for retrieving data and meta-data from URLs. An additional *context*

parameter, accepted by most stream creation functions, is passed to the wrapper's *stream_opener* method to fine-tune the behavior of the wrapper.

Any stream, once opened, can also have any number of *filters* applied to it, which process data as it is read from/written to the stream.

Streams can be cast (converted) into other types of file-handles, so that they can be used with third-party libraries without a great deal of trouble. This allows those libraries to access data directly from URL sources. If your system has the **fopencookie()** or **funopen()** function, you can even pass any PHP stream to any library that uses ANSI stdio!

Streams Basics

Using streams is very much like using ANSI stdio functions. The main difference is in how you obtain the stream handle to begin with. In most cases, you will use **php_stream_open_wrapper()** to obtain the stream handle. This function works very much like fopen, as can be seen from the example below:

Example #20 - simple stream example that displays the PHP home page

```
php_stream * stream = php_stream_open_wrapper("http://www.php.net", "rb",
REPORT_ERRORS, NULL);
if (stream) {
   while(!php_stream_eof(stream)) {
       char buf[1024];

       if (php_stream_gets(stream, buf, sizeof(buf))) {
           printf(buf);
       } else {
           break;
       }
   }
   php_stream_close(stream);
}
```

The table below shows the Streams equivalents of the more common ANSI stdio functions. Unless noted otherwise, the semantics of the functions are identical.

ANSI stdio equivalent functions in the Streams API

ANSI Stdio Function	PHP Streams Function	Notes
fopen	php_stream_open_wrapper	Streams includes additional parameters
fclose	php_stream_close	
fgets	php_stream_gets	

fread	php_stream_read	The nmemb parameter is assumed to have a value of 1, so the prototype looks more like read(2)
fwrite	php_stream_write	The nmemb parameter is assumed to have a value of 1, so the prototype looks more like write(2)
fseek	php_stream_seek	
ftell	php_stream_tell	
rewind	php_stream_rewind	
feof	php_stream_eof	
fgetc	php_stream_getc	
fputc	php_stream_putc	
fflush	php_stream_flush	
puts	php_stream_puts	Same semantics as puts, NOT fputs
fstat	php_stream_stat	Streams has a richer stat structure

Streams as Resources

All streams are registered as resources when they are created. This ensures that they will be properly cleaned up even if there is some fatal error. All of the filesystem functions in PHP operate on streams resources - that means that your extensions can accept regular PHP file pointers as parameters to, and return streams from their functions. The streams API makes this process as painless as possible:

Example #21 - How to accept a stream as a parameter

```
PHP_FUNCTION(example_write_hello)
{
   zval *zstream;
   php_stream *stream;

   if (FAILURE == zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r",
&zstream))
       return;
```

```
   php_stream_from_zval(stream, &zstream);

   /* you can now use the stream.  However, you do not "own" the
       stream, the script does.  That means you MUST NOT close the
       stream, because it will cause PHP to crash! */

   php_stream_write(stream, "hello\n");

   RETURN_TRUE();
}
```

Example #22 - How to return a stream from a function

```
PHP_FUNCTION(example_open_php_home_page)
{
   php_stream *stream;

   stream = php_stream_open_wrapper("http://www.php.net", "rb",
REPORT_ERRORS, NULL);

   php_stream_to_zval(stream, return_value);

   /* after this point, the stream is "owned" by the script.
       If you close it now, you will crash PHP! */
}
```

Since streams are automatically cleaned up, it's tempting to think that we can get away with being sloppy programmers and not bother to close the streams when we are done with them. Although such an approach might work, it is not a good idea for a number of reasons: streams hold locks on system resources while they are open, so leaving a file open after you have finished with it could prevent other processes from accessing it. If a script deals with a large number of files, the accumulation of the resources used, both in terms of memory and the sheer number of open files, can cause web server requests to fail. Sounds bad, doesn't it? The streams API includes some magic that helps you to keep your code clean - if a stream is not closed by your code when it should be, you will find some helpful debugging information in you web server error log.

Note

Always use a debug build of PHP when developing an extension (*--enable-debug* when running configure), as a lot of effort has been made to warn you about memory and stream leaks.

In some cases, it is useful to keep a stream open for the duration of a request, to act as a log or trace file for example. Writing the code to safely clean up such a stream is not difficult, but it's several lines of code that are not strictly needed. To save yourself the

trouble of writing the code, you can mark a stream as being OK for auto cleanup. What this means is that the streams API will not emit a warning when it is time to auto-cleanup a stream. To do this, you can use **php_stream_auto_cleanup()**.

Streams open options

These constants affect the operation of stream factory functions.

IGNORE_PATH
: This is the default option for streams; it requests that the include_path is not to be searched for the requested file.

USE_PATH
: Requests that the include_path is to be searched for the requested file.

IGNORE_URL
: Requests that registered URL wrappers are to be ignored when opening the stream. Other non-URL wrappers will be taken into consideration when decoding the path. There is no opposite form for this flag; the streams API will use all registered wrappers by default.

IGNORE_URL_WIN
: On Windows systems, this is equivalent to IGNORE_URL. On all other systems, this flag has no effect.

ENFORCE_SAFE_MODE
: Requests that the underlying stream implementation perform safe_mode checks on the file before opening the file. Omitting this flag will skip safe_mode checks and allow opening of any file that the PHP process has rights to access.

REPORT_ERRORS
: If this flag is set, and there was an error during the opening of the file or URL, the streams API will call the php_error function for you. This is useful because the path may contain username/password information that should not be displayed in the browser output (it would be a security risk to do so). When the streams API raises the error, it first strips username/password information from the path, making the error message safe to display in the browser.

STREAM_MUST_SEEK
: This flag is useful when your extension really must be able to randomly seek around in a stream. Some streams may not be seekable in their native form, so this flag asks the streams API to check to see if the stream does support seeking. If it does not, it will copy the stream into temporary storage (which may be a temporary file or a memory stream) which does support seeking. Please note that this flag is not useful when you want to seek the stream and write to it, because the stream you are accessing might not be bound to the actual resource you requested.

Note
If the requested resource is network based, this flag will cause the opener to block until the whole contents have been downloaded.

STREAM_WILL_CAST

If your extension is using a third-party library that expects a FILE* or file descriptor, you can use this flag to request the streams API to open the resource but avoid buffering. You can then use **php_stream_cast()** to retrieve the FILE* or file descriptor that the library requires. The is particularly useful when accessing HTTP URLs where the start of the actual stream data is found after an indeterminate offset into the stream. Since this option disables buffering at the streams API level, you may experience lower performance when using streams functions on the stream; this is deemed acceptable because you have told streams that you will be using the functions to match the underlying stream implementation. Only use this option when you are sure you need it.

Zend API: Hacking the Core of PHP

Introduction

Those who know don't talk.

Those who talk don't know.

Sometimes, PHP "as is" simply isn't enough. Although these cases are rare for the average user, professional applications will soon lead PHP to the edge of its capabilities, in terms of either speed or functionality. New functionality cannot always be implemented natively due to language restrictions and inconveniences that arise when having to carry around a huge library of default code appended to every single script, so another method needs to be found for overcoming these eventual lacks in PHP.

As soon as this point is reached, it's time to touch the heart of PHP and take a look at its core, the C code that makes PHP go.

Warning
This information is currently rather outdated, parts of it only cover early stages of the ZendEngine 1.0 API as it was used in early versions of PHP 4. More recent information may be found in the various README files that come with the PHP source and the » Internals section on the Zend website.

Overview

"Extending PHP" is easier said than done. PHP has evolved to a full-fledged tool consisting of a few megabytes of source code, and to hack a system like this quite a few things have to be learned and considered. When structuring this chapter, we finally decided on the "learn by doing" approach. This is not the most scientific and professional approach, but the method that's the most fun and gives the best end results. In the

following sections, you'll learn quickly how to get the most basic extensions to work almost instantly. After that, you'll learn about Zend's advanced API functionality. The alternative would have been to try to impart the functionality, design, tips, tricks, etc. as a whole, all at once, thus giving a complete look at the big picture before doing anything practical. Although this is the "better" method, as no dirty hacks have to be made, it can be very frustrating as well as energy- and time-consuming, which is why we've decided on the direct approach.

Note that even though this chapter tries to impart as much knowledge as possible about the inner workings of PHP, it's impossible to really give a complete guide to extending PHP that works 100% of the time in all cases. PHP is such a huge and complex package that its inner workings can only be understood if you make yourself familiar with it by practicing, so we encourage you to work with the source.

What Is Zend? and What Is PHP?

The name *Zend* refers to the language engine, PHP's core. The term *PHP* refers to the complete system as it appears from the outside. This might sound a bit confusing at first, but it's not that complicated (see below). To implement a Web script interpreter, you need three parts:

- The *interpreter* part analyzes the input code, translates it, and executes it.
- The *functionality* part implements the functionality of the language (its functions, etc.).
- The *interface* part talks to the Web server, etc.

Zend takes part 1 completely and a bit of part 2; PHP takes parts 2 and 3. Together they form the complete PHP package. Zend itself really forms only the language core, implementing PHP at its very basics with some predefined functions. PHP contains all the modules that actually create the language's outstanding capabilities. The internal structure of PHP.

The following sections discuss where PHP can be extended and how it's done.

Extension Possibilities

As shown above, PHP can be extended primarily at three points: external modules, built-in modules, and the Zend engine. The following sections discuss these options.

External Modules

External modules can be loaded at script runtime using the function dl(). This function loads a shared object from disk and makes its functionality available to the script to which it's being bound. After the script is terminated, the external module is discarded from memory. This method has both advantages and disadvantages, as described in the following table:

Advantages	Disadvantages
External modules don't require recompiling of PHP.	The shared objects need to be loaded every time a script is being executed (every hit), which is very slow.
The size of PHP remains small by "outsourcing" certain functionality.	External additional files clutter up the disk.
	Every script that wants to use an external module's functionality has to specifically include a call to dl(), or the *extension* tag in *php.ini* needs to be modified (which is not always a suitable solution).

To sum up, external modules are great for third-party products, small additions to PHP that are rarely used, or just for testing purposes. To develop additional functionality quickly, external modules provide the best results. For frequent usage, larger implementations, and complex code, the disadvantages outweigh the advantages.

Third parties might consider using the *extension* tag in *php.ini* to create additional external modules to PHP. These external modules are completely detached from the main package, which is a very handy feature in commercial environments. Commercial distributors can simply ship disks or archives containing only their additional modules, without the need to create fixed and solid PHP binaries that don't allow other modules to be bound to them.

Built-in Modules

Built-in modules are compiled directly into PHP and carried around with every PHP process; their functionality is instantly available to every script that's being run. Like external modules, built-in modules have advantages and disadvantages, as described in the following table:

Advantages	Disadvantages
No need to load the module specifically; the functionality is instantly available.	Changes to built-in modules require recompiling of PHP.
No external files clutter up the disk; everything resides in the PHP binary.	The PHP binary grows and consumes more memory.

Built-in modules are best when you have a solid library of functions that remains relatively unchanged, requires better than poor-to-average performance, or is used frequently by many scripts on your site. The need to recompile PHP is quickly compensated by the benefit in speed and ease of use. However, built-in modules are not ideal when rapid development of small additions is required.

The Zend Engine

Of course, extensions can also be implemented directly in the Zend engine. This strategy is good if you need a change in the language behavior or require special functions to be built directly into the language core. In general, however, modifications to the Zend engine should be avoided. Changes here result in incompatibilities with the rest of the world, and hardly anyone will ever adapt to specially patched Zend engines. Modifications can't be detached from the main PHP sources and are overridden with the next update using the "official" source repositories. Therefore, this method is generally considered bad practice and, due to its rarity, is not covered in this book.

Source Layout

Note
Prior to working through the rest of this chapter, you should retrieve clean, unmodified source trees of your favorite Web server. We're working with Apache (available at » http://www.apache.org/) and, of course, with PHP (available at » http://www.php.net/ - does it need to be said?). Make sure that you can compile a working PHP environment by yourself! We won't go into this issue here, however, as you should already have this most basic ability when studying this chapter.

Before we start discussing code issues, you should familiarize yourself with the source tree to be able to quickly navigate through PHP's files. This is a must-have ability to implement and debug extensions.

The following table describes the contents of the major directories.

Directory	Contents
php-src	Main PHP source files and main header files; here you'll find all of PHP's API definitions, macros, etc. (important). Everything else is below this directory.
php-src/ext	Repository for dynamic and built-in modules; by default, these are the "official" PHP modules that have been integrated into the main source tree. From PHP 4.0, it's possible to compile these standard extensions as dynamic loadable modules (at least, those that support it).
php-src/main	This directory contains the main php macros and definitions. (important)
php-src/pear	Directory for the PHP Extension and Application Repository. This directory

	contains core PEAR files.
php-src/sapi	Contains the code for the different server abstraction layers.
TSRM	Location of the "Thread Safe Resource Manager" (TSRM) for Zend and PHP.
ZendEngine2	Location of the Zend Engine files; here you'll find all of Zend's API definitions, macros, etc. (important).

Discussing all the files included in the PHP package is beyond the scope of this chapter. However, you should take a close look at the following files:

- *php-src/main/php.h*, located in the main PHP directory. This file contains most of PHP's macro and API definitions.
- *php-src/Zend/zend.h*, located in the main Zend directory. This file contains most of Zend's macros and definitions.
- *php-src/Zend/zend_API.h*, also located in the Zend directory, which defines Zend's API.

You should also follow some sub-inclusions from these files; for example, the ones relating to the Zend executor, the PHP initialization file support, and such. After reading these files, take the time to navigate around the package a little to see the interdependencies of all files and modules - how they relate to each other and especially how they make use of each other. This also helps you to adapt to the coding style in which PHP is authored. To extend PHP, you should quickly adapt to this style.

Extension Conventions

Zend is built using certain conventions; to avoid breaking its standards, you should follow the rules described in the following sections.

Macros

For almost every important task, Zend ships predefined macros that are extremely handy. The tables and figures in the following sections describe most of the basic functions, structures, and macros. The macro definitions can be found mainly in *zend.h* and *zend_API.h*. We suggest that you take a close look at these files after having studied this chapter. (Although you can go ahead and read them now, not everything will make sense to you yet.)

Memory Management

Resource management is a crucial issue, especially in server software. One of the most valuable resources is memory, and memory management should be handled with extreme

care. Memory management has been partially abstracted in Zend, and you should stick to this abstraction for obvious reasons: Due to the abstraction, Zend gets full control over all memory allocations. Zend is able to determine whether a block is in use, automatically freeing unused blocks and blocks with lost references, and thus prevent memory leaks. The functions to be used are described in the following table:

Function	Description
emalloc()	Serves as replacement for **malloc()**.
efree()	Serves as replacement for **free()**.
estrdup()	Serves as replacement for **strdup()**.
estrndup()	Serves as replacement for **strndup()**. Faster than **estrdup()** and binary-safe. This is the recommended function to use if you know the string length prior to duplicating it.
ecalloc()	Serves as replacement for **calloc()**.
erealloc()	Serves as replacement for **realloc()**.

emalloc(), **estrdup()**, **estrndup()**, **ecalloc()**, and **erealloc()** allocate internal memory; **efree()** frees these previously allocated blocks. Memory handled by the **e*()** functions is considered local to the current process and is discarded as soon as the script executed by this process is terminated.

Warning
To allocate resident memory that survives termination of the current script, you can use **malloc()** and **free()**. This should only be done with extreme care, however, and only in conjunction with demands of the Zend API; otherwise, you risk memory leaks.

Zend also features a thread-safe resource manager to provide better native support for multithreaded Web servers. This requires you to allocate local structures for all of your global variables to allow concurrent threads to be run. Because the thread-safe mode of Zend was not finished back when this was written, it is not yet extensively covered here.

Directory and File Functions

The following directory and file functions should be used in Zend modules. They behave exactly like their C counterparts, but provide virtual working directory support on the thread level.

Zend Function	Regular C Function
V_GETCWD()	getcwd()
V_FOPEN()	fopen()

V_OPEN()	**open()**
V_CHDIR()	chdir()
V_GETWD()	**getwd()**
V_CHDIR_FILE()	Takes a file path as an argument and changes the current working directory to that file's directory.
V_STAT()	stat()
V_LSTAT()	lstat()

String Handling

Strings are handled a bit differently by the Zend engine than other values such as integers, Booleans, etc., which don't require additional memory allocation for storing their values. If you want to return a string from a function, introduce a new string variable to the symbol table, or do something similar, you have to make sure that the memory the string will be occupying has previously been allocated, using the aforementioned **e*()** functions for allocation. (This might not make much sense to you yet; just keep it somewhere in your head for now - we'll get back to it shortly.)

Complex Types

Complex types such as arrays and objects require different treatment. Zend features a single API for these types - they're stored using hash tables.

Note
To reduce complexity in the following source examples, we're only working with simple types such as integers at first. A discussion about creating more advanced types follows later in this chapter.

PHP's Automatic Build System

PHP 4 features an automatic build system that's very flexible. All modules reside in a subdirectory of the *ext* directory. In addition to its own sources, each module consists of a config.m4 file, for extension configuration. (for example, see » http://www.gnu.org/software/m4/manual/m4.html)

All these stub files are generated automatically, along with*.cvsignore*, by a little shell script named *ext_skel* that resides in the *ext* directory. As argument it takes the name of the module that you want to create. The shell script then creates a directory of the same

name, along with the appropriate stub files.

Step by step, the process looks like this:

```
:~/cvs/php4/ext:> ./ext_skel --extname=my_module
Creating directory my_module
Creating basic files: config.m4 .cvsignore my_module.c php_my_module.h CREDITS
EXPERIMENTAL tests/001.phpt my_module.php [done].

To use your new extension, you will have to execute the following steps:

1.  $ cd ..
2.  $ vi ext/my_module/config.m4
3.  $ ./buildconf
4.  $ ./configure --[with|enable]-my_module
5.  $ make
6.  $ ./php -f ext/my_module/my_module.php
7.  $ vi ext/my_module/my_module.c
8.  $ make

Repeat steps 3-6 until you are satisfied with ext/my_module/config.m4 and
step 6 confirms that your module is compiled into PHP. Then, start writing
code and repeat the last two steps as often as necessary.
```

This instruction creates the aforementioned files. To include the new module in the automatic configuration and build process, you have to run *buildconf*, which regenerates the *configure* script by searching through the *ext* directory and including all found *config.m4* files.

The default *config.m4* shown in The default config.m4. is a bit more complex:

Example #23 - The default *config.m4*.

```
dnl $Id: build.xml,v 1.3 2007/11/01 16:40:36 rquadling Exp $
dnl config.m4 for extension my_module

dnl Comments in this file start with the string 'dnl'.
dnl Remove where necessary. This file will not work
dnl without editing.

dnl If your extension references something external, use with:

dnl PHP_ARG_WITH(my_module, for my_module support,
dnl Make sure that the comment is aligned:
dnl [  --with-my_module             Include my_module support])

dnl Otherwise use enable:

dnl PHP_ARG_ENABLE(my_module, whether to enable my_module support,
dnl Make sure that the comment is aligned:
dnl [  --enable-my_module           Enable my_module support])

if test "$PHP_MY_MODULE" != "no"; then
 dnl Write more examples of tests here...

 dnl # --with-my_module -> check with-path
 dnl SEARCH_PATH="/usr/local /usr"     # you might want to change this
 dnl SEARCH_FOR="/include/my_module.h"  # you most likely want to change
this
```

```
 dnl if test -r $PHP_MY_MODULE/; then # path given as parameter
 dnl   MY_MODULE_DIR=$PHP_MY_MODULE
 dnl else # search default path list
 dnl   AC_MSG_CHECKING([for my_module files in default path])
 dnl   for i in $SEARCH_PATH ; do
 dnl     if test -r $i/$SEARCH_FOR; then
 dnl       MY_MODULE_DIR=$i
 dnl       AC_MSG_RESULT(found in $i)
 dnl     fi
 dnl   done
 dnl fi
 dnl
 dnl if test -z "$MY_MODULE_DIR"; then
 dnl   AC_MSG_RESULT([not found])
 dnl   AC_MSG_ERROR([Please reinstall the my_module distribution])
 dnl fi

 dnl # --with-my_module -> add include path
 dnl PHP_ADD_INCLUDE($MY_MODULE_DIR/include)

 dnl # --with-my_module -> chech for lib and symbol presence
 dnl LIBNAME=my_module # you may want to change this
 dnl LIBSYMBOL=my_module # you most likely want to change this

 dnl PHP_CHECK_LIBRARY($LIBNAME,$LIBSYMBOL,
 dnl [
 dnl   PHP_ADD_LIBRARY_WITH_PATH($LIBNAME, $MY_MODULE_DIR/lib,
MY_MODULE_SHARED_LIBADD)
 dnl   AC_DEFINE(HAVE_MY_MODULELIB,1,[ ])
 dnl ],[
 dnl   AC_MSG_ERROR([wrong my_module lib version or lib not found])
 dnl ],[
 dnl   -L$MY_MODULE_DIR/lib -lm -ldl
 dnl ])
 dnl
 dnl PHP_SUBST(MY_MODULE_SHARED_LIBADD)

 PHP_NEW_EXTENSION(my_module, my_module.c, $ext_shared)
fi
```

If you're unfamiliar with M4 files (now is certainly a good time to get familiar), this might be a bit confusing at first; but it's actually quite easy.

Note: Everything prefixed with *dnl* is treated as a comment and is not parsed.

The *config.m4* file is responsible for parsing the command-line options passed to *configure* at configuration time. This means that it has to check for required external files and do similar configuration and setup tasks.

The default file creates two configuration directives in the *configure* script: *--with-my_module* and *--enable-my_module*. Use the first option when referring external files (such as the *--with-apache* directive that refers to the Apache directory). Use the second option when the user simply has to decide whether to enable your extension. Regardless of which option you use, you should uncomment the other, unnecessary one; that is, if you're using *--enable-my_module*, you should remove support for *--with-my_module*, and vice versa.

By default, the *config.m4* file created by *ext_skel* accepts both directives and automatically enables your extension. Enabling the extension is done by using the *PHP_EXTENSION* macro. To change the default behavior to include your module into the PHP binary when desired by the user (by explicitly specifying *--enable-my_module* or *--with-my_module*), change the test for *$PHP_MY_MODULE* to *== "yes"*:

```
if test "$PHP_MY_MODULE" == "yes"; then dnl
   Action.. PHP_EXTENSION(my_module, $ext_shared)
   fi
```

This would require you to use *--enable-my_module* each time when reconfiguring and recompiling PHP.

Note: Be sure to run *buildconf* every time you change *config.m4* !

We'll go into more details on the M4 macros available to your configuration scripts later in this chapter. For now, we'll simply use the default files.

Creating Extensions

We'll start with the creation of a very simple extension at first, which basically does nothing more than implement a function that returns the integer it receives as parameter. A simple extension. shows the source.

Example #24 - A simple extension.

```
/* include standard header */
#include "php.h"

/* declaration of functions to be exported */
ZEND_FUNCTION(first_module);

/* compiled function list so Zend knows what's in this module */
zend_function_entry firstmod_functions[] =
{
   ZEND_FE(first_module, NULL)
   {NULL, NULL, NULL}
};

/* compiled module information */
zend_module_entry firstmod_module_entry =
{
   STANDARD_MODULE_HEADER,
   "First Module",
   firstmod_functions,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   NO_VERSION_YET,
   STANDARD_MODULE_PROPERTIES
};

/* implement standard "stub" routine to introduce ourselves to Zend */
#if COMPILE_DL_FIRST_MODULE
ZEND_GET_MODULE(firstmod)
```

```
#endif

/* implement function that is meant to be made available to PHP */
ZEND_FUNCTION(first_module)
{
   long parameter;

   if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "l", &amp;parameter)
== FAILURE) {
       return;
   }

   RETURN_LONG(parameter);
}
```

This code contains a complete PHP module. We'll explain the source code in detail shortly, but first we'd like to discuss the build process. (This will allow the impatient to experiment before we dive into API discussions.)

Note
The example source makes use of some features introduced with the Zend version used in PHP 4.1.0 and above, it won't compile with older PHP 4.0.x versions.

Compiling Modules

There are basically two ways to compile modules:

- Use the provided "make" mechanism in the *ext* directory, which also allows building of dynamic loadable modules.
- Compile the sources manually.

The first method should definitely be favored, since, as of PHP 4.0, this has been standardized into a sophisticated build process. The fact that it is so sophisticated is also its drawback, unfortunately - it's hard to understand at first. We'll provide a more detailed introduction to this later in the chapter, but first let's work with the default files.

The second method is good for those who (for some reason) don't have the full PHP source tree available, don't have access to all files, or just like to juggle with their keyboard. These cases should be extremely rare, but for the sake of completeness we'll also describe this method.

Compiling Using Make

To compile the sample sources using the standard mechanism, copy all their subdirectories to the *ext* directory of your PHP source tree. Then run *buildconf*, which will create an updated *configure* script containing appropriate options for the new extension. By default, all the sample sources are disabled, so you don't have to fear breaking your build process.

After you run *buildconf*, *configure --help* shows the following additional modules:

```
--enable-array_experiments   BOOK: Enables array experiments
 --enable-call_userland       BOOK: Enables userland module
 --enable-cross_conversion    BOOK: Enables cross-conversion module
 --enable-first_module        BOOK: Enables first module
 --enable-infoprint           BOOK: Enables infoprint module
 --enable-reference_test      BOOK: Enables reference test module
 --enable-resource_test       BOOK: Enables resource test module
 --enable-variable_creation   BOOK: Enables variable-creation module
```

The module shown earlier in A simple extension. can be enabled with *--enable-first_module* or *--enable-first_module=yes*.

Compiling Manually

To compile your modules manually, you need the following commands:

Action	Command
Compiling	cc -fpic -DCOMPILE_DL_FIRST_MODULE=1 -I/usr/local/include -I. -I.. -I../Zend -c -o *<your_object_file> <your_c_file>*
Linking	cc -shared -L/usr/local/lib -rdynamic -o *<your_module_file> <your_object_file(s)>*

The command to compile the module simply instructs the compiler to generate position-independent code (*-fpic* shouldn't be omitted) and additionally defines the constant *COMPILE_DL_FIRST_MODULE* to tell the module code that it's compiled as a dynamically loadable module (the test module above checks for this; we'll discuss it shortly). After these options, it specifies a number of standard include paths that should be used as the minimal set to compile the source files.

Note: All include paths in the example are relative to the directory *ext*. If you're compiling from another directory, change the pathnames accordingly. Required items are the PHP directory, the *Zend* directory, and (if necessary), the directory in which your module resides.

The link command is also a plain vanilla command instructing linkage as a dynamic module.

You can include optimization options in the compilation command, although these have been omitted in this example (but some are included in the makefile template described in an earlier section).

Note: Compiling and linking manually as a static module into the PHP binary involves very long instructions and thus is not discussed here. (It's not very efficient to type all those commands.)

Using Extensions

Depending on the build process you selected, you should either end up with a new PHP binary to be linked into your Web server (or run as CGI), or with an .so (shared object) file. If you compiled the example file *first_module.c* as a shared object, your result file should be *first_module.so*. To use it, you first have to copy it to a place from which it's accessible to PHP. For a simple test procedure, you can copy it to your *htdocs* directory and try it with the source in A test file for first_module.so.. If you compiled it into the PHP binary, omit the call to dl(), as the module's functionality is instantly available to your scripts.

Warning

For security reasons, you *should not* put your dynamic modules into publicly accessible directories. Even though it *can* be done and it simplifies testing, you should put them into a separate directory in production environments.

Example #25 - A test file for first_module.so.

```
<?php

// remove next comment if necessary
// dl("first_module.so");

$param = 2;
$return = first_module($param);

print("We sent '$param' and got '$return'");

?>
```

Calling this PHP file should output the following:

```
We sent '2' and got '2'
```

If required, the dynamic loadable module is loaded by calling the dl() function. This function looks for the specified shared object, loads it, and makes its functions available to PHP. The module exports the function **first_module()**, which accepts a single parameter, converts it to an integer, and returns the result of the conversion.

If you've gotten this far, congratulations! You just built your first extension to PHP.

Troubleshooting

Actually, not much troubleshooting can be done when compiling static or dynamic modules. The only problem that could arise is that the compiler will complain about missing definitions or something similar. In this case, make sure that all header files are available and that you specified their path correctly in the compilation command. To be sure that everything is located correctly, extract a clean PHP source tree and use the automatic build in the *ext* directory with the fresh files; this will guarantee a safe

compilation environment. If this fails, try manual compilation.

PHP might also complain about missing functions in your module. (This shouldn't happen with the sample sources if you didn't modify them.) If the names of external functions you're trying to access from your module are misspelled, they'll remain as "unlinked symbols" in the symbol table. During dynamic loading and linkage by PHP, they won't resolve because of the typing errors - there are no corresponding symbols in the main binary. Look for incorrect declarations in your module file or incorrectly written external references. Note that this problem is specific to dynamic loadable modules; it doesn't occur with static modules. Errors in static modules show up at compile time.

Source Discussion

Now that you've got a safe build environment and you're able to include the modules into PHP files, it's time to discuss how everything works.

Module Structure

All PHP modules follow a common structure:

- Header file inclusions (to include all required macros, API definitions, etc.)
- C declaration of exported functions (required to declare the Zend function block)
- Declaration of the Zend function block
- Declaration of the Zend module block
- Implementation of **get_module()**
- Implementation of all exported functions

Header File Inclusions

The only header file you really have to include for your modules is *php.h*, located in the PHP directory. This file makes all macros and API definitions required to build new modules available to your code.

Tip: It's good practice to create a separate header file for your module that contains module-specific definitions. This header file should contain all the forward definitions for exported functions and include *php.h*. If you created your module using *ext_skel* you already have such a header file prepared.

Declaring Exported Functions

To declare functions that are to be exported (i.e., made available to PHP as new native functions), Zend provides a set of macros. A sample declaration looks like this:

```
ZEND_FUNCTION ( my_function );
```

ZEND_FUNCTION declares a new C function that complies with Zend's internal API. This means that the function is of type *void* and accepts *INTERNAL_FUNCTION_PARAMETERS* (another macro) as parameters. Additionally, it prefixes the function name with *zif*. The immediately expanded version of the above definitions would look like this:

```
void zif_my_function ( INTERNAL_FUNCTION_PARAMETERS );
```

Expanding *INTERNAL_FUNCTION_PARAMETERS* results in the following:

```
void zif_my_function( int ht
                    , zval * return_value
                    , zval * this_ptr
                    , int return_value_used
                    , zend_executor_globals * executor_globals
                    );
```

Since the interpreter and executor core have been separated from the main PHP package, a second API defining macros and function sets has evolved: the Zend API. As the Zend API now handles quite a few of the responsibilities that previously belonged to PHP, a lot of PHP functions have been reduced to macros aliasing to calls into the Zend API. The recommended practice is to use the Zend API wherever possible, as the old API is only preserved for compatibility reasons. For example, the types zval and pval are identical. zval is Zend's definition; pval is PHP's definition (actually, pval is an alias for zval now). As the macro *INTERNAL_FUNCTION_PARAMETERS* is a Zend macro, the above declaration contains zval. When writing code, you should always use zval to conform to the new Zend API.

The parameter list of this declaration is very important; you should keep these parameters in mind (see Zend's Parameters to Functions Called from PHP for descriptions).

Zend's Parameters to Functions Called from PHP

Parameter	Description
ht	The number of arguments passed to the Zend function. You should not touch this directly, but instead use ZEND_NUM_ARGS() to obtain the value.
return_value	This variable is used to pass any return values of your function back to PHP. Access to this variable is best done using the predefined macros. For a description of these see below.
this_ptr	Using this variable, you can gain access to the object in which your function is contained, if it's used within an object. Use the function **getThis()** to obtain this pointer.
return_value_used	This flag indicates whether an eventual return value from this function will actually be used by the calling script. *0* indicates that the return value is not used; *1* indicates that

	the caller expects a return value. Evaluation of this flag can be done to verify correct usage of the function as well as speed optimizations in case returning a value requires expensive operations (for an example, see how *array.c* makes use of this).
executor_globals	This variable points to global settings of the Zend engine. You'll find this useful when creating new variables, for example (more about this later). The executor globals can also be introduced to your function by using the macro *TSRMLS_FETCH()*.

Declaration of the Zend Function Block

Now that you have declared the functions to be exported, you also have to introduce them to Zend. Introducing the list of functions is done by using an array of zend_function_entry. This array consecutively contains all functions that are to be made available externally, with the function's name as it should appear in PHP and its name as defined in the C source. Internally, zend_function_entry is defined as shown in Internal declaration of zend_function_entry..

Example #26 - Internal declaration of zend_function_entry.

```
typedef struct _zend_function_entry {
   char *fname;
   void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
   unsigned char *func_arg_types;
} zend_function_entry;
```

Entry	Description
fname	Denotes the function name as seen in PHP (for example, *fopen*, *mysql_connect*, or, in our example, *first_module*).
handler	Pointer to the C function responsible for handling calls to this function. For example, see the standard macro *INTERNAL_FUNCTION_PARAMETERS* discussed earlier.
func_arg_types	Allows you to mark certain parameters so that they're forced to be passed by reference. You usually should set this to

	NULL.

In the example above, the declaration looks like this:

```
zend_function_entry firstmod_functions[] =
{
   ZEND_FE(first_module, NULL)
   {NULL, NULL, NULL}
};
```

You can see that the last entry in the list always has to be *{NULL, NULL, NULL}*. This marker has to be set for Zend to know when the end of the list of exported functions is reached.

Note
You *cannot* use the predefined macros for the end marker, as these would try to refer to a function named "NULL"!

The macro *ZEND_FE* (short for 'Zend Function Entry') simply expands to a structure entry in zend_function_entry. Note that these macros introduce a special naming scheme to your functions - your C functions will be prefixed with *zif_*, meaning that *ZEND_FE(first_module)* will refer to a C function **zif_first_module()**. If you want to mix macro usage with hand-coded entries (not a good practice), keep this in mind.

Tip: Compilation errors that refer to functions named **zif_*()** relate to functions defined with *ZEND_FE*.

Macros for Defining Functions shows a list of all the macros that you can use to define functions.

Macros for Defining Functions

Macro Name	Description
ZEND_FE(name, arg_types)	Defines a function entry of the name name in zend_function_entry. Requires a corresponding C function. arg_types needs to be set to *NULL*. This function uses automatic C function name generation by prefixing the PHP function name with *zif_*. For example, *ZEND_FE("first_module", NULL)* introduces a function **first_module()** to PHP and links it to the C function **zif_first_module()**. Use in conjunction with *ZEND_FUNCTION*.
ZEND_NAMED_FE(php_name, name, arg_types)	Defines a function that will be available to PHP by the name php_name and links it to the corresponding C function name. arg_types needs to be set to *NULL*. Use this function if you don't want the automatic

	name prefixing introduced by *ZEND_FE*. Use in conjunction with *ZEND_NAMED_FUNCTION*.
ZEND_FALIAS(name, alias, arg_types)	Defines an alias named alias for name. arg_types needs to be set to *NULL*. Doesn't require a corresponding C function; refers to the alias target instead.
PHP_FE(name, arg_types)	Old PHP API equivalent of *ZEND_FE*.
PHP_NAMED_FE(runtime_name, name, arg_types)	Old PHP API equivalent of *ZEND_NAMED_FE*.

Note: You can't use *ZEND_FE* in conjunction with *PHP_FUNCTION*, or *PHP_FE* in conjunction with *ZEND_FUNCTION*. However, it's perfectly legal to mix *ZEND_FE* and *ZEND_FUNCTION* with *PHP_FE* and *PHP_FUNCTION* when staying with the same macro set for each function to be declared. But mixing is *not* recommended; instead, you're advised to use the *ZEND_** macros only.

Declaration of the Zend Module Block

This block is stored in the structure zend_module_entry and contains all necessary information to describe the contents of this module to Zend. You can see the internal definition of this module in Internal declaration of zend_module_entry..

Example #27 - Internal declaration of zend_module_entry.

```
typedef struct _zend_module_entry zend_module_entry;

   struct _zend_module_entry {
   unsigned short size;
   unsigned int zend_api;
   unsigned char zend_debug;
   unsigned char zts;
   char *name;
   zend_function_entry *functions;
   int (*module_startup_func)(INIT_FUNC_ARGS);
   int (*module_shutdown_func)(SHUTDOWN_FUNC_ARGS);
   int (*request_startup_func)(INIT_FUNC_ARGS);
   int (*request_shutdown_func)(SHUTDOWN_FUNC_ARGS);
   void (*info_func)(ZEND_MODULE_INFO_FUNC_ARGS);
   char *version;

[ Rest of the structure is not interesting here ]

};
```

Entry	Description
size, zend_api, zend_debug and zts	Usually filled with the

	"STANDARD_MODULE_HEADER", which fills these four members with the size of the whole zend_module_entry, the *ZEND_MODULE_API_NO*, whether it is a debug build or normal build (*ZEND_DEBUG*) and if ZTS is enabled (*USING_ZTS*).
name	Contains the module name (for example, *"File functions"*, *"Socket functions"*, *"Crypt"*, etc.). This name will show up in phpinfo(), in the section "Additional Modules."
functions	Points to the Zend function block, discussed in the preceding section.
module_startup_func	This function is called once upon module initialization and can be used to do one-time initialization steps (such as initial memory allocation, etc.). To indicate a failure during initialization, return *FAILURE*; otherwise, *SUCCESS*. To mark this field as unused, use *NULL*. To declare a function, use the macro *ZEND_MINIT*.
module_shutdown_func	This function is called once upon module shutdown and can be used to do one-time deinitialization steps (such as memory deallocation). This is the counterpart to **module_startup_func()**. To indicate a failure during deinitialization, return *FAILURE*; otherwise, *SUCCESS*. To mark this field as unused, use *NULL*. To declare a function, use the macro *ZEND_MSHUTDOWN*.
request_startup_func	This function is called once upon every page request and can be used to do one-time initialization steps that are required to process a request. To indicate a failure here, return *FAILURE*; otherwise, *SUCCESS*. *Note:* As dynamic loadable modules are loaded only on page requests, the request startup function is called right after the module startup function (both initialization events happen at the same time). To mark this field as unused, use *NULL*. To declare a function, use the macro *ZEND_RINIT*.
request_shutdown_func	This function is called once after every page request and works as counterpart to

	request_startup_func(). To indicate a failure here, return *FAILURE*; otherwise, *SUCCESS*. *Note:* As dynamic loadable modules are loaded only on page requests, the request shutdown function is immediately followed by a call to the module shutdown handler (both deinitialization events happen at the same time). To mark this field as unused, use *NULL*. To declare a function, use the macro *ZEND_RSHUTDOWN*.
info_func	When phpinfo() is called in a script, Zend cycles through all loaded modules and calls this function. Every module then has the chance to print its own "footprint" into the output page. Generally this is used to dump environmental or statistical information. To mark this field as unused, use *NULL*. To declare a function, use the macro *ZEND_MINFO*.
version	The version of the module. You can use *NO_VERSION_YET* if you don't want to give the module a version number yet, but we really recommend that you add a version string here. Such a version string can look like this (in chronological order): *"2.5-dev"*, *"2.5RC1"*, *"2.5"* or *"2.5pl3"*.
Remaining structure elements	These are used internally and can be prefilled by using the macro *STANDARD_MODULE_PROPERTIES_EX*. You should not assign any values to them. Use *STANDARD_MODULE_PROPERTIES_EX* only if you use global startup and shutdown functions; otherwise, use *STANDARD_MODULE_PROPERTIES* directly.

In our example, this structure is implemented as follows:

```
zend_module_entry firstmod_module_entry =
{
   STANDARD_MODULE_HEADER,
   "First Module",
   firstmod_functions,
   NULL, NULL, NULL, NULL, NULL,
   NO_VERSION_YET,
   STANDARD_MODULE_PROPERTIES,
};
```

This is basically the easiest and most minimal set of values you could ever use. The module name is set to *First Module*, then the function list is referenced, after which all startup and shutdown functions are marked as being unused.

For reference purposes, you can find a list of the macros involved in declared startup and shutdown functions in Macros to Declare Startup and Shutdown Functions. These are not used in our basic example yet, but will be demonstrated later on. You should make use of these macros to declare your startup and shutdown functions, as these require special arguments to be passed (*INIT_FUNC_ARGS* and *SHUTDOWN_FUNC_ARGS*), which are automatically included into the function declaration when using the predefined macros. If you declare your functions manually and the PHP developers decide that a change in the argument list is necessary, you'll have to change your module sources to remain compatible.

Macros to Declare Startup and Shutdown Functions

Macro	Description
ZEND_MINIT(module)	Declares a function for module startup. The generated name will be *zend_minit_<module>* (for example, *zend_minit_first_module*). Use in conjunction with *ZEND_MINIT_FUNCTION*.
ZEND_MSHUTDOWN(module)	Declares a function for module shutdown. The generated name will be *zend_mshutdown_<module>* (for example, *zend_mshutdown_first_module*). Use in conjunction with *ZEND_MSHUTDOWN_FUNCTION*.
ZEND_RINIT(module)	Declares a function for request startup. The generated name will be *zend_rinit_<module>* (for example, *zend_rinit_first_module*). Use in conjunction with *ZEND_RINIT_FUNCTION*.
ZEND_RSHUTDOWN(module)	Declares a function for request shutdown. The generated name will be *zend_rshutdown_<module>* (for example, *zend_rshutdown_first_module*). Use in conjunction with *ZEND_RSHUTDOWN_FUNCTION*.
ZEND_MINFO(module)	Declares a function for printing module information, used when phpinfo() is called. The generated name will be *zend_info_<module>* (for example, *zend_info_first_module*). Use in conjunction with *ZEND_MINFO_FUNCTION*.

Creation of get_module()

This function is special to all dynamic loadable modules. Take a look at the creation via the *ZEND_GET_MODULE* macro first:

```
#if COMPILE_DL_FIRSTMOD
    ZEND_GET_MODULE(firstmod)
#endif
```

The function implementation is surrounded by a conditional compilation statement. This is needed since the function **get_module()** is only required if your module is built as a dynamic extension. By specifying a definition of *COMPILE_DL_FIRSTMOD* in the compiler command (see above for a discussion of the compilation instructions required to build a dynamic extension), you can instruct your module whether you intend to build it as a dynamic extension or as a built-in module. If you want a built-in module, the implementation of **get_module()** is simply left out.

get_module() is called by Zend at load time of the module. You can think of it as being invoked by the dl() call in your script. Its purpose is to pass the module information block back to Zend in order to inform the engine about the module contents.

If you don't implement a **get_module()** function in your dynamic loadable module, Zend will compliment you with an error message when trying to access it.

Implementation of All Exported Functions

Implementing the exported functions is the final step. The example function in *first_module* looks like this:

```
ZEND_FUNCTION(first_module)
{
   long parameter;

   if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "l", &amp;parameter) ==
FAILURE) {
       return;
   }

   RETURN_LONG(parameter);
}
```

The function declaration is done using *ZEND_FUNCTION*, which corresponds to *ZEND_FE* in the function entry table (discussed earlier).

After the declaration, code for checking and retrieving the function's arguments, argument conversion, and return value generation follows (more on this later).

Summary

That's it, basically - there's nothing more to implementing PHP modules. Built-in modules are structured similarly to dynamic modules, so, equipped with the information presented in the previous sections, you'll be able to fight the odds when encountering PHP module source files.

Now, in the following sections, read on about how to make use of PHP's internals to build powerful extensions.

Accepting Arguments

One of the most important issues for language extensions is accepting and dealing with data passed via arguments. Most extensions are built to deal with specific input data (or require parameters to perform their specific actions), and function arguments are the only real way to exchange data between the PHP level and the C level. Of course, there's also the possibility of exchanging data using predefined global values (which is also discussed later), but this should be avoided by all means, as it's extremely bad practice.

PHP doesn't make use of any formal function declarations; this is why call syntax is always completely dynamic and never checked for errors. Checking for correct call syntax is left to the user code. For example, it's possible to call a function using only one argument at one time and four arguments the next time - both invocations are syntactically absolutely correct.

Determining the Number of Arguments

Since PHP doesn't have formal function definitions with support for call syntax checking, and since PHP features variable arguments, sometimes you need to find out with how many arguments your function has been called. You can use the *ZEND_NUM_ARGS* macro in this case. In previous versions of PHP, this macro retrieved the number of arguments with which the function has been called based on the function's hash table entry, ht, which is passed in the *INTERNAL_FUNCTION_PARAMETERS* list. As ht itself now contains the number of arguments that have been passed to the function, *ZEND_NUM_ARGS* has been stripped down to a dummy macro (see its definition in *zend_API.h*). But it's still good practice to use it, to remain compatible with future changes in the call interface. *Note:* The old PHP equivalent of this macro is *ARG_COUNT*.

The following code checks for the correct number of arguments:

```
if(ZEND_NUM_ARGS() != 2) WRONG_PARAM_COUNT;
```

If the function is not called with two arguments, it exits with an error message. The code snippet above makes use of the tool macro *WRONG_PARAM_COUNT*, which can be used to generate a standard error message like:"Warning: Wrong parameter count for firstmodule() in /home/www/htdocs/firstmod.php on line 5"

This macro prints a default error message and then returns to the caller. Its definition can also be found in *zend_API.h* and looks like this:

```
ZEND_API void wrong_param_count(void);

#define WRONG_PARAM_COUNT { wrong_param_count(); return; }
```

As you can see, it calls an internal function named **wrong_param_count()** that's responsible for printing the warning. For details on generating customized error messages, see the later section "Printing Information."

Retrieving Arguments

Note

New parameter parsing API

This chapter documents the new Zend parameter parsing API introduced by Andrei Zmievski. It was introduced in the development stage between PHP 4.0.6 and 4.1.0 .

Parsing parameters is a very common operation and it may get a bit tedious. It would also be nice to have standardized error checking and error messages. Since PHP 4.1.0, there is a way to do just that by using the new parameter parsing API. It greatly simplifies the process of receiving parameters, but it has a drawback in that it can't be used for functions that expect variable number of parameters. But since the vast majority of functions do not fall into those categories, this parsing API is recommended as the new standard way.

The prototype for parameter parsing function looks like this:

```
int zend_parse_parameters(int num_args TSRMLS_DC, char *type_spec, ...);
```

The first argument to this function is supposed to be the number of actual parameters passed to your function, so *ZEND_NUM_ARGS()* can be used for that. The second parameter should always be *TSRMLS_CC* macro. The third argument is a string that specifies the number and types of arguments your function is expecting, similar to how printf format string specifies the number and format of the output values it should operate on. And finally the rest of the arguments are pointers to variables which should receive the values from the parameters.

zend_parse_parameters() also performs type conversions whenever possible, so that you always receive the data in the format you asked for. Any type of scalar can be converted to another one, but conversions between complex types (arrays, objects, and resources) and scalar types are not allowed.

If the parameters could be obtained successfully and there were no errors during type conversion, the function will return *SUCCESS*, otherwise it will return *FAILURE*. The function will output informative error messages, if the number of received parameters does not match the requested number, or if type conversion could not be performed.

Here are some sample error messages:

```
Warning - ini_get_all() requires at most 1 parameter, 2 given
Warning - wddx_deserialize() expects parameter 1 to be string, array given
```

Of course each error message is accompanied by the filename and line number on which it occurs.

Here is the full list of type specifiers:

- *l* - long
- *d* - double
- *s* - string (with possible null bytes) and its length
- *b* - boolean
- *r* - resource, stored in *zval**

- *a* - array, stored in *zval**
- *o* - object (of any class), stored in *zval**
- *O* - object (of class specified by class entry), stored in *zval**
- *z* - the actual *zval**

The following characters also have a meaning in the specifier string:

- | - indicates that the remaining parameters are optional. The storage variables corresponding to these parameters should be initialized to default values by the extension, since they will not be touched by the parsing function if the parameters are not passed.
- */* - the parsing function will call **SEPARATE_ZVAL_IF_NOT_REF()** on the parameter it follows, to provide a copy of the parameter, unless it's a reference.
- *!* - the parameter it follows can be of specified type or *NULL* (only applies to a, o, O, r, and z). If *NULL* value is passed by the user, the storage pointer will be set to *NULL*.

The best way to illustrate the usage of this function is through examples:

```
/* Gets a long, a string and its length, and a zval. */
long l;
char *s;
int s_len;
zval *param;
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
                          "lsz", &l, &s, &s_len, &param) == FAILURE) {
   return;
}

/* Gets an object of class specified by my_ce, and an optional double. */
zval *obj;
double d = 0.5;
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
                          "O|d", &obj, my_ce, &d) == FAILURE) {
   return;
}

/* Gets an object or null, and an array.
  If null is passed for object, obj will be set to NULL. */
zval *obj;
zval *arr;
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "O!a", &obj, &arr) ==
FAILURE) {
   return;
}

/* Gets a separated array. */
zval *arr;
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a/", &arr) == FAILURE) {
   return;
}

/* Get only the first three parameters (useful for varargs functions). */
zval *z;
zend_bool b;
```

```
zval *r;
if (zend_parse_parameters(3, "zbr!", &z, &b, &r) == FAILURE) {
   return;
}
```

Note that in the last example we pass 3 for the number of received parameters, instead of **ZEND_NUM_ARGS()**. What this lets us do is receive the least number of parameters if our function expects a variable number of them. Of course, if you want to operate on the rest of the parameters, you will have to use **zend_get_parameters_array_ex()** to obtain them.

The parsing function has an extended version that allows for an additional flags argument that controls its actions.

```
int zend_parse_parameters_ex(int flags, int num_args TSRMLS_DC, char *type_spec,
...);
```

The only flag you can pass currently is *ZEND_PARSE_PARAMS_QUIET*, which instructs the function to not output any error messages during its operation. This is useful for functions that expect several sets of completely different arguments, but you will have to output your own error messages.

For example, here is how you would get either a set of three longs or a string:

```
long l1, l2, l3;
char *s;
if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET,
                             ZEND_NUM_ARGS() TSRMLS_CC,
                             "lll", &l1, &l2, &l3) == SUCCESS) {
   /* manipulate longs */
} else if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET,
                                    ZEND_NUM_ARGS(), "s", &s, &s_len) == SUCCESS)
{
   /* manipulate string */
} else {
   php_error(E_WARNING, "%s() takes either three long values or a string as
argument",
             get_active_function_name(TSRMLS_C));
   return;
}
```

With all the abovementioned ways of receiving function parameters you should have a good handle on this process. For even more example, look through the source code for extensions that are shipped with PHP - they illustrate every conceivable situation.

Old way of retrieving arguments (deprecated)

Note
Deprecated parameter parsing API This API is deprecated and superseded by the new ZEND parameter parsing API.

After having checked the number of arguments, you need to get access to the arguments themselves. This is done with the help of **zend_get_parameters_ex()**:

```
zval **parameter;

if(zend_get_parameters_ex(1, &parameter) != SUCCESS)
 WRONG_PARAM_COUNT;
```

All arguments are stored in a zval container, which needs to be pointed to *twice*. The snippet above tries to retrieve one argument and make it available to us via the parameter pointer.

zend_get_parameters_ex() accepts at least two arguments. The first argument is the number of arguments to retrieve (which should match the number of arguments with which the function has been called; this is why it's important to check for correct call syntax). The second argument (and all following arguments) are pointers to pointers to pointers to zval s. (Confusing, isn't it?) All these pointers are required because Zend works internally with **zval; to adjust a local **zval in our function, **zend_get_parameters_ex()** requires a pointer to it.

The return value of **zend_get_parameters_ex()** can either be *SUCCESS* or *FAILURE*, indicating (unsurprisingly) success or failure of the argument processing. A failure is most likely related to an incorrect number of arguments being specified, in which case you should exit with *WRONG_PARAM_COUNT*.

To retrieve more than one argument, you can use a similar snippet:

```
zval **param1, **param2, **param3, **param4;

if(zend_get_parameters_ex(4, &param1, &param2, &param3, &param4) != SUCCESS)
   WRONG_PARAM_COUNT;
```

zend_get_parameters_ex() only checks whether you're trying to retrieve too many parameters. If the function is called with five arguments, but you're only retrieving three of them with **zend_get_parameters_ex()**, you won't get an error but will get the first three parameters instead. Subsequent calls of **zend_get_parameters_ex()** won't retrieve the remaining arguments, but will get the same arguments again.

Dealing with a Variable Number of Arguments/Optional Parameters

If your function is meant to accept a variable number of arguments, the snippets just described are sometimes suboptimal solutions. You have to create a line calling **zend_get_parameters_ex()** for every possible number of arguments, which is often unsatisfying.

For this case, you can use the function **zend_get_parameters_array_ex()**, which accepts the number of parameters to retrieve and an array in which to store them:

```
zval **parameter_array[4];

/* get the number of arguments */
argument_count = ZEND_NUM_ARGS();

/* see if it satisfies our minimal request (2 arguments) */
/* and our maximal acceptance (4 arguments) */
if(argument_count < 2 || argument_count > 4)
```

```
   WRONG_PARAM_COUNT;

/* argument count is correct, now retrieve arguments */
if(zend_get_parameters_array_ex(argument_count, parameter_array) != SUCCESS)
   WRONG_PARAM_COUNT;
```

First, the number of arguments is checked to make sure that it's in the accepted range. After that, **zend_get_parameters_array_ex()** is used to fill parameter_array with valid pointers to the argument values.

A very clever implementation of this can be found in the code handling PHP's fsockopen() located in *ext/standard/fsock.c*, as shown in PHP's implementation of variable arguments in fsockopen().. Don't worry if you don't know all the functions used in this source yet; we'll get to them shortly.

Example #28 - PHP's implementation of variable arguments in fsockopen().

```
pval **args[5];
int *sock=emalloc(sizeof(int));
int *sockp;
int arg_count=ARG_COUNT(ht);
int socketd = -1;
unsigned char udp = 0;
struct timeval timeout = { 60, 0 };
unsigned short portno;
unsigned long conv;
char *key = NULL;
FLS_FETCH();

if (arg_count > 5 || arg_count < 2 ||
zend_get_parameters_array_ex(arg_count,args)==FAILURE) {
   CLOSE_SOCK(1);
   WRONG_PARAM_COUNT;
}

switch(arg_count) {
   case 5:
       convert_to_double_ex(args[4]);
       conv = (unsigned long) (Z_DVAL_PP(args[4]) * 1000000.0);
       timeout.tv_sec = conv / 1000000;
       timeout.tv_usec = conv % 1000000;
       /* fall-through */
   case 4:
       if (!PZVAL_IS_REF(*args[3])) {
           php_error(E_WARNING,"error string argument to fsockopen not
passed by reference");
       }
       pval_copy_constructor(*args[3]);
       ZVAL_EMPTY_STRING(*args[3]);
       /* fall-through */
   case 3:
       if (!PZVAL_IS_REF(*args[2])) {
           php_error(E_WARNING,"error argument to fsockopen not passed by
reference");
           return;
       }
       ZVAL_LONG(*args[2], 0);
       break;
}
```

```
convert_to_string_ex(args[0]);
convert_to_long_ex(args[1]);
portno = (unsigned short) Z_LVAL_P(args[1]);

key = emalloc(Z_STRLEN_P(args[0]) + 10);
```

fsockopen() accepts two, three, four, or five parameters. After the obligatory variable declarations, the function checks for the correct range of arguments. Then it uses a fall-through mechanism in a *switch()* statement to deal with all arguments. The *switch()* statement starts with the maximum number of arguments being passed (five). After that, it automatically processes the case of four arguments being passed, then three, by omitting the otherwise obligatory *break* keyword in all stages. After having processed the last case, it exits the *switch()* statement and does the minimal argument processing needed if the function is invoked with only two arguments.

This multiple-stage type of processing, similar to a stairway, allows convenient processing of a variable number of arguments.

Accessing Arguments

To access arguments, it's necessary for each argument to have a clearly defined type. Again, PHP's extremely dynamic nature introduces some quirks. Because PHP never does any kind of type checking, it's possible for a caller to pass any kind of data to your functions, whether you want it or not. If you expect an integer, for example, the caller might pass an array, and vice versa - PHP simply won't notice.

To work around this, you have to use a set of API functions to force a type conversion on every argument that's being passed (see Argument Conversion Functions).

Note: All conversion functions expect a **zval as parameter.

Argument Conversion Functions

Function	Description
convert_to_boolean_ex()	Forces conversion to a Boolean type. Boolean values remain untouched. Longs, doubles, and strings containing *0* as well as NULL values will result in Boolean *0* (FALSE). Arrays and objects are converted based on the number of entries or properties, respectively, that they have. Empty arrays and objects are converted to FALSE; otherwise, to TRUE. All other values result in a Boolean *1* (TRUE).
convert_to_long_ex()	Forces conversion to a long, the default integer type. NULL values, Booleans, resources, and of course longs remain

	untouched. Doubles are truncated. Strings containing an integer are converted to their corresponding numeric representation, otherwise resulting in *0*. Arrays and objects are converted to *0* if empty, *1* otherwise.
convert_to_double_ex()	Forces conversion to a double, the default floating-point type. NULL values, Booleans, resources, longs, and of course doubles remain untouched. Strings containing a number are converted to their corresponding numeric representation, otherwise resulting in *0.0*. Arrays and objects are converted to *0.0* if empty, *1.0* otherwise.
convert_to_string_ex()	Forces conversion to a string. Strings remain untouched. NULL values are converted to an empty string. Booleans containing TRUE are converted to *"1"*, otherwise resulting in an empty string. Longs and doubles are converted to their corresponding string representation. Arrays are converted to the string *"Array"* and objects to the string *"Object"*.
convert_to_array_ex(value)	Forces conversion to an array. Arrays remain untouched. Objects are converted to an array by assigning all their properties to the array table. All property names are used as keys, property contents as values. NULL values are converted to an empty array. All other values are converted to an array that contains the specific source value in the element with the key *0*.
convert_to_object_ex(value)	Forces conversion to an object. Objects remain untouched. NULL values are converted to an empty object. Arrays are converted to objects by introducing their keys as properties into the objects and their values as corresponding property contents in the object. All other types result in an object with the property *scalar*, having the corresponding source value as content.
convert_to_null_ex(value)	Forces the type to become a NULL value, meaning empty.

Note

You can find a demonstration of the behavior in *cross_conversion.php* on the accompanying CD-ROM.

Cross-conversion behavior of PHP.

Using these functions on your arguments will ensure type safety for all data that's passed to you. If the supplied type doesn't match the required type, PHP forces dummy contents on the resulting value (empty strings, arrays, or objects, *0* for numeric values, *FALSE* for Booleans) to ensure a defined state.

Following is a quote from the sample module discussed previously, which makes use of the conversion functions:

```
zval **parameter;

if((ZEND_NUM_ARGS() != 1) || (zend_get_parameters_ex(1, &parameter) != SUCCESS))
{
   WRONG_PARAM_COUNT;
}

convert_to_long_ex(parameter);

RETURN_LONG(Z_LVAL_P(parameter));
```

After retrieving the parameter pointer, the parameter value is converted to a long (an integer), which also forms the return value of this function. Understanding access to the contents of the value requires a short discussion of the zval type, whose definition is shown in PHP/Zend zval type definition..

Example #29 - PHP/Zend zval type definition.

```
typedef pval zval;

typedef struct _zval_struct zval;

typedef union _zvalue_value {
   long lval;                  /* long value */
   double dval;                /* double value */
   struct {
       char *val;
       int len;
   } str;
   HashTable *ht;              /* hash table value */
   struct {
       zend_class_entry *ce;
       HashTable *properties;
   } obj;
} zvalue_value;

struct _zval_struct {
   /* Variable information */
   zvalue_value value;         /* value */
   unsigned char type;         /* active type */
   unsigned char is_ref;
```

```
    short refcount;
};
```

Actually, pval (defined in *php.h*) is only an alias of zval (defined in *zend.h*), which in turn refers to _zval_struct. This is a most interesting structure. _zval_struct is the "master" structure, containing the value structure, type, and reference information. The substructure zvalue_value is a union that contains the variable's contents. Depending on the variable's type, you'll have to access different members of this union. For a description of both structures, see Zend zval Structure, Zend zvalue_value Structure and Zend Variable Type Constants.

Zend zval Structure

Entry	Description
value	Union containing this variable's contents. See Zend zvalue_value Structure for a description.
type	Contains this variable's type. For a list of available types, see Zend Variable Type Constants.
is_ref	0 means that this variable is not a reference; 1 means that this variable is a reference to another variable.
refcount	The number of references that exist for this variable. For every new reference to the value stored in this variable, this counter is increased by 1. For every lost reference, this counter is decreased by 1. When the reference counter reaches 0, no references exist to this value anymore, which causes automatic freeing of the value.

Zend zvalue_value Structure

Entry	Description
lval	Use this property if the variable is of the type *IS_LONG*, *IS_BOOLEAN*, or *IS_RESOURCE*.
dval	Use this property if the variable is of the type *IS_DOUBLE*.
str	This structure can be used to access

	variables of the type *IS_STRING*. The member len contains the string length; the member val points to the string itself. Zend uses C strings; thus, the string length contains a trailing *0x00*.
ht	This entry points to the variable's hash table entry if the variable is an array.
obj	Use this property if the variable is of the type *IS_OBJECT*.

Zend Variable Type Constants

Constant	Description
IS_NULL	Denotes a NULL (empty) value.
IS_LONG	A long (integer) value.
IS_DOUBLE	A double (floating point) value.
IS_STRING	A string.
IS_ARRAY	Denotes an array.
IS_OBJECT	An object.
IS_BOOL	A Boolean value.
IS_RESOURCE	A resource (for a discussion of resources, see the appropriate section below).
IS_CONSTANT	A constant (defined) value.

To access a long you access zval.value.lval, to access a double you use zval.value.dval, and so on. Because all values are stored in a union, trying to access data with incorrect union members results in meaningless output.

Accessing arrays and objects is a bit more complicated and is discussed later.

Dealing with Arguments Passed by Reference

If your function accepts arguments passed by reference that you intend to modify, you need to take some precautions.

What we didn't say yet is that under the circumstances presented so far, you don't have write access to any zval containers designating function parameters that have been

passed to you. Of course, you can change any zval containers that you created within your function, but you mustn't change any zval s that refer to Zend-internal data!

We've only discussed the so-called ***_ex()** API so far. You may have noticed that the API functions we've used are called **zend_get_parameters_ex()** instead of **zend_get_parameters()**, **convert_to_long_ex()** instead of **convert_to_long()**, etc. The ***_ex()** functions form the so-called new "extended" Zend API. They give a minor speed increase over the old API, but as a tradeoff are only meant for providing read-only access.

Because Zend works internally with references, different variables may reference the same value. Write access to a zval container requires this container to contain an isolated value, meaning a value that's not referenced by any other containers. If a zval container were referenced by other containers and you changed the referenced zval, you would automatically change the contents of the other containers referencing this zval (because they'd simply point to the changed value and thus change their own value as well).

zend_get_parameters_ex() doesn't care about this situation, but simply returns a pointer to the desired zval containers, whether they consist of references or not. Its corresponding function in the traditional API, **zend_get_parameters()**, immediately checks for referenced values. If it finds a reference, it creates a new, isolated zval container; copies the referenced data into this newly allocated space; and then returns a pointer to the new, isolated value.

This action is called *zval separation* (or pval separation). Because the ***_ex()** API doesn't perform zval separation, it's considerably faster, while at the same time disabling write access.

To change parameters, however, write access is required. Zend deals with this situation in a special way: Whenever a parameter to a function is passed by reference, it performs automatic zval separation. This means that whenever you're calling a function like this in PHP, Zend will automatically ensure that $parameter is being passed as an isolated value, rendering it to a write-safe state:

```
my_function(&amp;$parameter);
```

But this *is not* the case with regular parameters! All other parameters that are not passed by reference are in a read-only state.

This requires you to make sure that you're really working with a reference - otherwise you might produce unwanted results. To check for a parameter being passed by reference, you can use the macro *PZVAL_IS_REF*. This macro accepts a *zval** to check if it is a reference or not. Examples are given in in Testing for referenced parameter passing..

Example #30 - Testing for referenced parameter passing.

```
zval *parameter;

if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &parameter) ==
FAILURE)
   return;

/* check for parameter being passed by reference */
```

```
if (!PZVAL_IS_REF(parameter)) {
{
   zend_error(E_WARNING, "Parameter wasn't passed by reference");
   RETURN_NULL();
}

/* make changes to the parameter */
ZVAL_LONG(parameter, 10);
```

Assuring Write Safety for Other Parameters

You might run into a situation in which you need write access to a parameter that's retrieved with **zend_get_parameters_ex()** but not passed by reference. For this case, you can use the macro *SEPARATE_ZVAL*, which does a zval separation on the provided container. The newly generated zval is detached from internal data and has only a local scope, meaning that it can be changed or destroyed without implying global changes in the script context:

```
zval **parameter;

/* retrieve parameter */
zend_get_parameters_ex(1, &parameter);

/* at this stage, <parameter> still is connected */
/* to Zend's internal data buffers */

/* make <parameter> write-safe */
SEPARATE_ZVAL(parameter);

/* now we can safely modify <parameter> */
/* without implying global changes */
```

SEPARATE_ZVAL uses **emalloc()** to allocate the new zval container, which means that even if you don't deallocate this memory yourself, it will be destroyed automatically upon script termination. However, doing a lot of calls to this macro without freeing the resulting containers will clutter up your RAM.

Note: As you can easily work around the lack of write access in the "traditional" API (with **zend_get_parameters()** and so on), this API seems to be obsolete, and is not discussed further in this chapter.

Creating Variables

When exchanging data from your own extensions with PHP scripts, one of the most important issues is the creation of variables. This section shows you how to deal with the variable types that PHP supports.

Overview

To create new variables that can be seen "from the outside" by the executing script, you need to allocate a new zval container, fill this container with meaningful values, and then

introduce it to Zend's internal symbol table. This basic process is common to all variable creations:

```
zval *new_variable;

/* allocate and initialize new container */
MAKE_STD_ZVAL(new_variable);

/* set type and variable contents here, see the following sections */

/* introduce this variable by the name "new_variable_name" into the symbol table
*/
ZEND_SET_SYMBOL(EG(active_symbol_table), "new_variable_name", new_variable);

/* the variable is now accessible to the script by using $new_variable_name */
```

The macro *MAKE_STD_ZVAL* allocates a new zval container using *ALLOC_ZVAL* and initializes it using *INIT_ZVAL*. As implemented in Zend at the time of this writing, *initializing* means setting the reference count to *1* and clearing the is_ref flag, but this process could be extended later - this is why it's a good idea to keep using *MAKE_STD_ZVAL* instead of only using *ALLOC_ZVAL*. If you want to optimize for speed (and you don't have to explicitly initialize the zval container here), you can use *ALLOC_ZVAL*, but this isn't recommended because it doesn't ensure data integrity.

ZEND_SET_SYMBOL takes care of introducing the new variable to Zend's symbol table. This macro checks whether the value already exists in the symbol table and converts the new symbol to a reference if so (with automatic deallocation of the old zval container). This is the preferred method if speed is not a crucial issue and you'd like to keep memory usage low.

Note that *ZEND_SET_SYMBOL* makes use of the Zend executor globals via the macro *EG*. By specifying *EG(active_symbol_table)*, you get access to the currently active symbol table, dealing with the active, local scope. The local scope may differ depending on whether the function was invoked from within a function.

If you need to optimize for speed and don't care about optimal memory usage, you can omit the check for an existing variable with the same value and instead force insertion into the symbol table by using **zend_hash_update()**:

```
zval *new_variable;

/* allocate and initialize new container */
MAKE_STD_ZVAL(new_variable);

/* set type and variable contents here, see the following sections */

/* introduce this variable by the name "new_variable_name" into the symbol table
*/
zend_hash_update(
   EG(active_symbol_table),
   "new_variable_name",
   strlen("new_variable_name") + 1,
   &amp;new_variable,
   sizeof(zval *),
   NULL
);
```

This is actually the standard method used in most modules.

The variables generated with the snippet above will always be of local scope, so they reside in the context in which the function has been called. To create new variables in the global scope, use the same method but refer to another symbol table:

```
zval *new_variable;

// allocate and initialize new container
MAKE_STD_ZVAL(new_variable);

//
// set type and variable contents here
//

// introduce this variable by the name "new_variable_name" into the global
symbol table
ZEND_SET_SYMBOL(&amp;EG(symbol_table), "new_variable_name", new_variable);
```

The macro *ZEND_SET_SYMBOL* is now being called with a reference to the main, global symbol table by referring *EG(symbol_table)*.

Note: The active_symbol_table variable is a pointer, but symbol_table is not. This is why you have to use *EG(active_symbol_table)* and *&EG(symbol_table)* as parameters to *ZEND_SET_SYMBOL* - it requires a pointer.

Similarly, to get a more efficient version, you can hardcode the symbol table update:

```
zval *new_variable;

// allocate and initialize new container
MAKE_STD_ZVAL(new_variable);

//
// set type and variable contents here
//

// introduce this variable by the name "new_variable_name" into the global
symbol table
zend_hash_update(
   &amp;EG(symbol_table),
   "new_variable_name",
   strlen("new_variable_name") + 1,
   &amp;new_variable,
   sizeof(zval *),
   NULL
);
```

Creating variables with different scopes. shows a sample source that creates two variables - local_variable with a local scope and global_variable with a global scope (see Figure 9.7). The full example can be found on the CD-ROM.

Note: You can see that the global variable is actually not accessible from within the function. This is because it's not imported into the local scope using *global $global_variable;* in the PHP source.

Example #31 - Creating variables with different scopes.

```
ZEND_FUNCTION(variable_creation)
{
   zval *new_var1, *new_var2;
```

```
    MAKE_STD_ZVAL(new_var1);
    MAKE_STD_ZVAL(new_var2);

    ZVAL_LONG(new_var1, 10);
    ZVAL_LONG(new_var2, 5);

    ZEND_SET_SYMBOL(EG(active_symbol_table), "local_variable", new_var1);
    ZEND_SET_SYMBOL(&amp;EG(symbol_table), "global_variable", new_var2);

    RETURN_NULL();

}
```

Longs (Integers)

Now let's get to the assignment of data to variables, starting with longs. Longs are PHP's integers and are very simple to store. Looking at the zval.value container structure discussed earlier in this chapter, you can see that the long data type is directly contained in the union, namely in the lval field. The corresponding type value for longs is *IS_LONG* (see Creation of a long.).

Example #32 - Creation of a long.

```
zval *new_long;

MAKE_STD_ZVAL(new_long);

new_long-&gt;type = IS_LONG;
new_long-&gt;value.lval = 10;
```

Alternatively, you can use the macro *ZVAL_LONG*:

```
zval *new_long;

MAKE_STD_ZVAL(new_long);
ZVAL_LONG(new_long, 10);
```

Doubles (Floats)

Doubles are PHP's floats and are as easy to assign as longs, because their value is also contained directly in the union. The member in the zval.value container is dval; the corresponding type is *IS_DOUBLE*.

```
zval *new_double;

MAKE_STD_ZVAL(new_double);

new_double-&gt;type = IS_DOUBLE;
new_double-&gt;value.dval = 3.45;
```

Alternatively, you can use the macro *ZVAL_DOUBLE*:

```
zval *new_double;

MAKE_STD_ZVAL(new_double);
```

```
ZVAL_DOUBLE(new_double, 3.45);
```

Strings

Strings need slightly more effort. As mentioned earlier, all strings that will be associated with Zend's internal data structures need to be allocated using Zend's own memory-management functions. Referencing of static strings or strings allocated with standard routines is not allowed. To assign strings, you have to access the structure str in the zval.value container. The corresponding type is *IS_STRING*:

```
zval *new_string;
char *string_contents = "This is a new string variable";

MAKE_STD_ZVAL(new_string);

new_string-&gt;type = IS_STRING;
new_string-&gt;value.str.len = strlen(string_contents);
new_string-&gt;value.str.val = estrdup(string_contents);
   </programlisting>
   Note the usage of Zend's <function>estrdup</function> here.
   Of course, you can also use the predefined macro
   <literal>ZVAL_STRING</literal>:
   <programlisting>
zval *new_string;
char *string_contents = "This is a new string variable";

MAKE_STD_ZVAL(new_string);
ZVAL_STRING(new_string, string_contents, 1);
```

ZVAL_STRING accepts a third parameter that indicates whether the supplied string contents should be duplicated (using **estrdup()**). Setting this parameter to *1* causes the string to be duplicated; *0* simply uses the supplied pointer for the variable contents. This is most useful if you want to create a new variable referring to a string that's already allocated in Zend internal memory.

If you want to truncate the string at a certain position or you already know its length, you can use *ZVAL_STRINGL(zval, string, length, duplicate)*, which accepts an explicit string length to be set for the new string. This macro is faster than *ZVAL_STRING* and also binary-safe.

To create empty strings, set the string length to *0* and use *empty_string* as contents:

```
new_string-&gt;type = IS_STRING;
new_string-&gt;value.str.len = 0;
new_string-&gt;value.str.val = empty_string;
```

Of course, there's a macro for this as well (*ZVAL_EMPTY_STRING*):

```
MAKE_STD_ZVAL(new_string);
ZVAL_EMPTY_STRING(new_string);
```

Booleans

Booleans are created just like longs, but have the type *IS_BOOL*. Allowed values in lval are *0* and *1*:

```
zval *new_bool;
```

```
MAKE_STD_ZVAL(new_bool);

new_bool-&gt;type = IS_BOOL;
new_bool-&gt;value.lval = 1;
```
The corresponding macros for this type are *ZVAL_BOOL* (allowing specification of the value) as well as *ZVAL_TRUE* and *ZVAL_FALSE* (which explicitly set the value to *TRUE* and *FALSE*, respectively).

Arrays

Arrays are stored using Zend's internal hash tables, which can be accessed using the **zend_hash_*()** API. For every array that you want to create, you need a new hash table handle, which will be stored in the ht member of the zval.value container.

There's a whole API solely for the creation of arrays, which is extremely handy. To start a new array, you call **array_init()**.
```
zval *new_array;

MAKE_STD_ZVAL(new_array);

array_init(new_array);
```
array_init() always returns *SUCCESS*.

To add new elements to the array, you can use numerous functions, depending on what you want to do. Zend's API for Associative Arrays, Zend's API for Indexed Arrays, Part 1 and Zend's API for Indexed Arrays, Part 2 describe these functions. All functions return *FAILURE* on failure and *SUCCESS* on success.

Zend's API for Associative Arrays

Function	Description
add_assoc_long(zval *array, char *key, long n);()	Adds an element of type *long*.
add_assoc_unset(zval *array, char *key);()	Adds an unset element.
add_assoc_bool(zval *array, char *key, int b);()	Adds a Boolean element.
add_assoc_resource(zval *array, char *key, int r);()	Adds a resource to the array.
add_assoc_double(zval *array, char *key, double d);()	Adds a floating-point value.
add_assoc_string(zval *array, char *key, char *str, int duplicate);()	Adds a string to the array. The flag duplicate specifies whether the string contents have to be copied to Zend internal memory.

add_assoc_stringl(zval *array, char *key, char *str, uint length, int duplicate);()	Adds a string with the desired length length to the array. Otherwise, behaves like **add_assoc_string()**.
add_assoc_zval(zval *array, char *key, zval *value);()	Adds a zval to the array. Useful for adding other arrays, objects, streams, etc...

Zend's API for Indexed Arrays, Part 1

Function	Description
add_index_long(zval *array, uint idx, long n);()	Adds an element of type *long*.
add_index_unset(zval *array, uint idx);()	Adds an unset element.
add_index_bool(zval *array, uint idx, int b);()	Adds a Boolean element.
add_index_resource(zval *array, uint idx, int r);()	Adds a resource to the array.
add_index_double(zval *array, uint idx, double d);()	Adds a floating-point value.
add_index_string(zval *array, uint idx, char *str, int duplicate);()	Adds a string to the array. The flag duplicate specifies whether the string contents have to be copied to Zend internal memory.
add_index_stringl(zval *array, uint idx, char *str, uint length, int duplicate);()	Adds a string with the desired length length to the array. This function is faster and binary-safe. Otherwise, behaves like **add_index_string()**.
add_index_zval(zval *array, uint idx, zval *value);()	Adds a zval to the array. Useful for adding other arrays, objects, streams, etc...

Zend's API for Indexed Arrays, Part 2

Function	Description
add_next_index_long(zval *array, long n);()	Adds an element of type *long*.
add_next_index_unset(zval *array);()	Adds an unset element.
add_next_index_bool(zval *array, int b);()	Adds a Boolean element.

add_next_index_resource(zval *array, int r);()	Adds a resource to the array.
add_next_index_double(zval *array, double d);()	Adds a floating-point value.
add_next_index_string(zval *array, char *str, int duplicate);()	Adds a string to the array. The flag duplicate specifies whether the string contents have to be copied to Zend internal memory.
add_next_index_stringl(zval *array, char *str, uint length, int duplicate);()	Adds a string with the desired length length to the array. This function is faster and binary-safe. Otherwise, behaves like **add_index_string()**.
add_next_index_zval(zval *array, zval *value);()	Adds a zval to the array. Useful for adding other arrays, objects, streams, etc...

All these functions provide a handy abstraction to Zend's internal hash API. Of course, you can also use the hash functions directly - for example, if you already have a zval container allocated that you want to insert into an array. This is done using **zend_hash_update()** for associative arrays (see Adding an element to an associative array.) and **zend_hash_index_update()** for indexed arrays (see Adding an element to an indexed array.):

Example #33 - Adding an element to an associative array.

```
zval *new_array, *new_element;
char *key = "element_key";

MAKE_STD_ZVAL(new_array);
MAKE_STD_ZVAL(new_element);

array_init(new_array);

ZVAL_LONG(new_element, 10);

if(zend_hash_update(new_array-&gt;value.ht, key, strlen(key) + 1, (void
*)&amp;new_element, sizeof(zval *), NULL) == FAILURE)
{
   // do error handling here
}
```

Example #34 - Adding an element to an indexed array.

```
zval *new_array, *new_element;
int key = 2;

MAKE_STD_ZVAL(new_array);
MAKE_STD_ZVAL(new_element);

array_init(new_array);
```

```
ZVAL_LONG(new_element, 10);

if(zend_hash_index_update(new_array-&gt;value.ht, key, (void
*)&amp;new_element, sizeof(zval *), NULL) == FAILURE)
{
   // do error handling here
}
```

To emulate the functionality of **add_next_index_*()**, you can use this:

```
zend_hash_next_index_insert(ht, zval **new_element, sizeof(zval *), NULL)
```

Note: To return arrays from a function, use **array_init()** and all following actions on the predefined variable return_value (given as argument to your exported function; see the earlier discussion of the call interface). You do not have to use *MAKE_STD_ZVAL* on this.

Tip: To avoid having to write *new_array->value.ht* every time, you can use *HASH_OF(new_array)*, which is also recommended for compatibility and style reasons.

Objects

Since objects can be converted to arrays (and vice versa), you might have already guessed that they have a lot of similarities to arrays in PHP. Objects are maintained with the same hash functions, but there's a different API for creating them.

To initialize an object, you use the function **object_init()**:

```
zval *new_object;

MAKE_STD_ZVAL(new_object);

if(object_init(new_object) != SUCCESS)
{
   // do error handling here
}
```

You can use the functions described in Zend's API for Object Creation to add members to your object.

Zend's API for Object Creation

Function	Description
add_property_long(zval *object, char *key, long l);()	Adds a long to the object.
add_property_unset(zval *object, char *key);()	Adds an unset property to the object.
add_property_bool(zval *object, char *key, int b);()	Adds a Boolean to the object.

add_property_resource(zval *object, char *key, long r);()	Adds a resource to the object.
add_property_double(zval *object, char *key, double d);()	Adds a double to the object.
add_property_string(zval *object, char *key, char *str, int duplicate);()	Adds a string to the object.
add_property_stringl(zval *object, char *key, char *str, uint length, int duplicate);()	Adds a string of the specified length to the object. This function is faster than **add_property_string()** and also binary-safe.
add_property_zval(zval *obect, char *key, zval *container):()	Adds a *zval* container to the object. This is useful if you have to add properties which aren't simple types like integers or strings but arrays or other objects.

Resources

Resources are a special kind of data type in PHP. The term *resources* doesn't really refer to any special kind of data, but to an abstraction method for maintaining any kind of information. Resources are kept in a special resource list within Zend. Each entry in the list has a correspondending type definition that denotes the kind of resource to which it refers. Zend then internally manages all references to this resource. Access to a resource is never possible directly - only via a provided API. As soon as all references to a specific resource are lost, a corresponding shutdown function is called.

For example, resources are used to store database links and file descriptors. The *de facto* standard implementation can be found in the MySQL module, but other modules such as the Oracle module also make use of resources.

Note
In fact, a resource can be a pointer to anything you need to handle in your functions (e.g. pointer to a structure) and the user only has to pass a single resource variable to your function.

To create a new resource you need to register a resource destruction handler for it. Since you can store any kind of data as a resource, Zend needs to know how to free this resource if its not longer needed. This works by registering your own resource destruction handler to Zend which in turn gets called by Zend whenever your resource can be freed (whether manually or automatically). Registering your resource handler within Zend returns you the *resource type handle* for that resource. This handle is needed whenever you want to access a resource of this type later and is most of time stored in a global static variable within your extension. There is no need to worry about thread safety here because you only register your resource handler once during module initialization.

The Zend function to register your resource handler is defined as:

```
ZEND_API int zend_register_list_destructors_ex(rsrc_dtor_func_t ld,
rsrc_dtor_func_t pld, char *type_name, int module_number);
```

There are two different kinds of resource destruction handlers you can pass to this function: a handler for normal resources and a handler for persistent resources. Persistent resources are for example used for database connection. When registering a resource, either of these handlers must be given. For the other handler just pass *NULL*.

zend_register_list_destructors_ex() accepts the following parameters:

ld	Normal resource destruction handler callback
pld	Pesistent resource destruction handler callback
type_name	A string specifying the name of your resource. It's always a good thing to specify a unique name within PHP for the resource type so when the user for example calls *var_dump($resource);* he also gets the name of the resource.
module_number	The *module_number* is automatically available in your *PHP_MINIT_FUNCTION* function and therefore you just pass it over.

The return value is a unique integer ID for your *resource type*.

The resource destruction handler (either normal or persistent resources) has the following prototype:

```
void resource_destruction_handler(zend_rsrc_list_entry *rsrc TSRMLS_DC);
```

The passed *rsrc* is a pointer to the following structure:

```
typedef struct _zend_rsrc_list_entry {

   void *ptr;
   int type;
   int refcount;

} zend_rsrc_list_entry;
```

The member *void *ptr* is the actual pointer to your resource.

Now we know how to start things, we define our own resource we want register within Zend. It is only a simple structure with two integer members:

```
typedef struct {

   int resource_link;
   int resource_type;

} my_resource;
```

Our resource destruction handler is probably going to look something like this:

```
void my_destruction_handler(zend_rsrc_list_entry *rsrc TSRMLS_DC) {
```

```
   // You most likely cast the void pointer to your structure type

   my_resource *my_rsrc = (my_resource *) rsrc->ptr;

   // Now do whatever needs to be done with you resource. Closing
   // Files, Sockets, freeing additional memory, etc.
   // Also, don't forget to actually free the memory for your resource too!

   do_whatever_needs_to_be_done_with_the_resource(my_rsrc);
}
```

Note
One important thing to mention: If your resource is a rather complex structure which also contains pointers to memory you allocated during runtime you have to free them *before* freeing the resource itself!

Now that we have defined

- what our resource is and
- our resource destruction handler

we can go on and do the rest of the steps:

- create a global variable within the extension holding the resource ID so it can be accessed from every function which needs it
- define the resource name
- write the resource destruction handler
- and finally register the handler

```
// Somewhere in your extension, define the variable for your registered
resources.
   // If you wondered what 'le' stands for: it simply means 'list entry'.
   static int le_myresource;

   // It's nice to define your resource name somewhere
   #define le_myresource_name  "My type of resource"

   [...]

   // Now actually define our resource destruction handler
   void my_destruction_handler(zend_rsrc_list_entry *rsrc TSRMLS_DC) {

       my_resource *my_rsrc = (my_resource *) rsrc->ptr;
       do_whatever_needs_to_be_done_with_the_resource(my_rsrc);
   }

   [...]

   PHP_MINIT_FUNCTION(my_extension) {
```

```
       // Note that 'module_number' is already provided through the
       // PHP_MINIT_FUNCTION() function definition.

       le_myresource = zend_register_list_destructors_ex(my_destruction_handler,
NULL, le_myresource_name, module_number);

       // You can register additional resources, initialize
       // your global vars, constants, whatever.
   }
```

To actually register a new resource you use can either use the **zend_register_resource()** function or the **ZEND_REGISTER_RESOURE()** macro, both defined in zend_list.h . Although the arguments for both map 1:1 it's a good idea to always use macros to be upwards compatible:

```
int ZEND_REGISTER_RESOURCE(zval *rsrc_result, void *rsrc_pointer, int
rsrc_type);
```

rsrc_result	This is an already initialized *zval* * container.
rsrc_pointer	Your resource pointer you want to store.
rsrc_type	The type which you received when you registered the resource destruction handler. If you followed the naming scheme this would be *le_myresource*.

The return value is a unique integer identifier for that resource.

What is really going on when you register a new resource is it gets inserted in an internal list in Zend and the result is just stored in the given *zval* * container:

```
rsrc_id = zend_list_insert(rsrc_pointer, rsrc_type);

   if (rsrc_result) {
       rsrc_result->value.lval = rsrc_id;
       rsrc_result->type = IS_RESOURCE;
   }

   return rsrc_id;
```

The returned *rsrc_id* uniquely identifies the newly registered resource. You can use the macro *RETURN_RESOURE* to return it to the user:

```
   RETURN_RESOURCE(rsrc_id)
```

Note
It is common practice that if you want to return the resource immediately to the user you specify the *return_value* as the *zval* * container.

Zend now keeps track of all references to this resource. As soon as all references to the resource are lost, the destructor that you previously registered for this resource is called. The nice thing about this setup is that you don't have to worry about memory leakages

introduced by allocations in your module - just register all memory allocations that your calling script will refer to as resources. As soon as the script decides it doesn't need them anymore, Zend will find out and tell you.

Now that the user got his resource, at some point he is passing it back to one of your functions. The value.lval inside the *zval* * container contains the key to your resource and thus can be used to fetch the resource with the following macro:
ZEND_FETCH_RESOURCE:

```
ZEND_FETCH_RESOURCE(rsrc, rsrc_type, rsrc_id, default_rsrc_id,
resource_type_name, resource_type)
```

rsrc	This is your pointer which will point to your previously registered resource.
rsrc_type	This is the typecast argument for your pointer, e.g. *myresource* *.
rsrc_id	This is the address of the *zval* * container the user passed to your function, e.g. *&z_resource* if *zval *z_resource* is given.
default_rsrc_id	This integer specifies the default resource *ID* if no resource could be fetched or -1.
resource_type_name	This is the name of the requested resource. It's a string and is used when the resource can't be found or is invalid to form a meaningful error message.
resource_type	The *resource_type* you got back when registering the resource destruction handler. In our example this was le_myresource.

This macro has no return value. It is for the developers convenience and takes care of TSRMLS arguments passing and also does check if the resource could be fetched. It throws a warning message and returns the current PHP function with *NULL* if there was a problem retrieving the resource.

To force removal of a resource from the list, use the function **zend_list_delete()**. You can also force the reference count to increase if you know that you're creating another reference for a previously allocated value (for example, if you're automatically reusing a default database link). For this case, use the function **zend_list_addref()**. To search for previously allocated resource entries, use **zend_list_find()**. The complete API can be found in *zend_list.h*.

Macros for Automatic Global Variable Creation

In addition to the macros discussed earlier, a few macros allow easy creation of simple global variables. These are nice to know in case you want to introduce global flags, for example. This is somewhat bad practice, but Table Macros for Global Variable Creation

describes macros that do exactly this task. They don't need any zval allocation; you simply have to supply a variable name and value.

Macros for Global Variable Creation

Macro	Description
SET_VAR_STRING(name, value)	Creates a new string.
SET_VAR_STRINGL(name, value, length)	Creates a new string of the specified length. This macro is faster than *SET_VAR_STRING* and also binary-safe.
SET_VAR_LONG(name, value)	Creates a new long.
SET_VAR_DOUBLE(name, value)	Creates a new double.

Creating Constants

Zend supports the creation of true constants (as opposed to regular variables). Constants are accessed without the typical dollar sign (*$*) prefix and are available in all scopes. Examples include *TRUE* and *FALSE*, to name just two.

To create your own constants, you can use the macros in Macros for Creating Constants. All the macros create a constant with the specified name and value.

You can also specify flags for each constant:

- *CONST_CS* - This constant's name is to be treated as case sensitive.
- *CONST_PERSISTENT* - This constant is persistent and won't be "forgotten" when the current process carrying this constant shuts down.

To use the flags, combine them using a inary OR:

```
// register a new constant of type "long"
    REGISTER_LONG_CONSTANT("NEW_MEANINGFUL_CONSTANT", 324, CONST_CS |
    CONST_PERSISTENT);
```

There are two types of macros - *REGISTER_*_CONSTANT* and *REGISTER_MAIN_*_CONSTANT*. The first type creates constants bound to the current module. These constants are dumped from the symbol table as soon as the module that registered the constant is unloaded from memory. The second type creates constants that remain in the symbol table independently of the module.

Macros for Creating Constants

Macro	Description
REGISTER_LONG_CONSTANT(name, value, flags)	*Registers a new constant of type long.*

REGISTER_MAIN_LONG_CONSTANT(name, value, flags)	
REGISTER_DOUBLE_CONSTANT(name, value, flags) *REGISTER_MAIN_DOUBLE_CONSTANT(name, value, flags)*	Registers a new constant of type double.
REGISTER_STRING_CONSTANT(name, value, flags) *REGISTER_MAIN_STRING_CONSTANT(name, value, flags)*	Registers a new constant of type string. The specified string must reside in Zend's internal memory.
REGISTER_STRINGL_CONSTANT(name, value, length, flags) *REGISTER_MAIN_STRINGL_CONSTANT(name, value, length, flags)*	Registers a new constant of type string. The string length is explicitly set to length. The specified string must reside in Zend's internal memory.

Duplicating Variable Contents: The Copy Constructor

Sooner or later, you may need to assign the contents of one zval container to another. This is easier said than done, since the zval container doesn't contain only type information, but also references to places in Zend's internal data. For example, depending on their size, arrays and objects may be nested with lots of hash table entries. By assigning one zval to another, you avoid duplicating the hash table entries, using only a reference to them (at most).

To copy this complex kind of data, use the *copy constructor*. Copy constructors are typically defined in languages that support operator overloading, with the express purpose of copying complex types. If you define an object in such a language, you have the possibility of overloading the "=" operator, which is usually responsible for assigning the contents of the rvalue (result of the evaluation of the right side of the operator) to the lvalue (same for the left side).

Overloading means assigning a different meaning to this operator, and is usually used to assign a function call to an operator. Whenever this operator would be used on such an object in a program, this function would be called with the lvalue and rvalue as parameters. Equipped with that information, it can perform the operation it intends the "=" operator to have (usually an extended form of copying).

This same form of "extended copying" is also necessary for PHP's zval containers. Again, in the case of an array, this extended copying would imply re-creation of all hash table entries relating to this array. For strings, proper memory allocation would have to be assured, and so on.

Zend ships with such a function, called **zend_copy_ctor()** (the previous PHP equivalent was **pval_copy_constructor()**).

A most useful demonstration is a function that accepts a complex type as argument, modifies it, and then returns the argument:

```
zval *parameter;

if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &amp;parameter) ==
FAILURE)
  return;
}

// do modifications to the parameter here

// now we want to return the modified container:
*return_value = *parameter;
zval_copy_ctor(return_value);
```

The first part of the function is plain-vanilla argument retrieval. After the (left out) modifications, however, it gets interesting: The container of parameter is assigned to the (predefined) return_value container. Now, in order to effectively duplicate its contents, the copy constructor is called. The copy constructor works directly with the supplied argument, and the standard return values are *FAILURE* on failure and *SUCCESS* on success.

If you omit the call to the copy constructor in this example, both parameter and return_value would point to the same internal data, meaning that return_value would be an illegal additional reference to the same data structures. Whenever changes occurred in the data that parameter points to, return_value might be affected. Thus, in order to create separate copies, the copy constructor must be used.

The copy constructor's counterpart in the Zend API, the destructor **zval_dtor()**, does the opposite of the constructor.

Returning Values

Returning values from your functions to PHP was described briefly in an earlier section; this section gives the details. Return values are passed via the return_value variable, which is passed to your functions as argument. The return_value argument consists of a zval container (see the earlier discussion of the call interface) that you can freely modify. The container itself is already allocated, so you don't have to run *MAKE_STD_ZVAL* on it. Instead, you can access its members directly.

To make returning values from functions easier and to prevent hassles with accessing the internal structures of the zval container, a set of predefined macros is available (as usual). These macros automatically set the correspondent type and value, as described in Predefined Macros for Returning Values from a Function and Predefined Macros for Setting the Return Value of a Function.

Note
The macros in Predefined Macros for Returning Values from a Function automatically *return* from your function, those in Predefined Macros for Setting the Return Value of a Function only *set* the return value; they don't return from your function.

Predefined Macros for Returning Values from a Function

Macro	Description
RETURN_RESOURCE(resource)	Returns a resource.
RETURN_BOOL(bool)	Returns a Boolean.
RETURN_NULL()	Returns nothing (a NULL value).
RETURN_LONG(long)	Returns a long.
RETURN_DOUBLE(double)	Returns a double.
RETURN_STRING(string, duplicate)	Returns a string. The duplicate flag indicates whether the string should be duplicated using **estrdup()**.
RETURN_STRINGL(string, length, duplicate)	Returns a string of the specified length; otherwise, behaves like *RETURN_STRING*. This macro is faster and binary-safe, however.
RETURN_EMPTY_STRING()	Returns an empty string.
RETURN_FALSE	Returns Boolean false.
RETURN_TRUE	Returns Boolean true.

Predefined Macros for Setting the Return Value of a Function

Macro	Description
RETVAL_RESOURCE(resource)	Sets the return value to the specified resource.
RETVAL_BOOL(bool)	Sets the return value to the specified Boolean value.
RETVAL_NULL	Sets the return value to NULL.
RETVAL_LONG(long)	Sets the return value to the specified long.
RETVAL_DOUBLE(double)	Sets the return value to the specified double.
RETVAL_STRING(string, duplicate)	Sets the return value to the specified string and duplicates it to Zend internal memory if desired (see also *RETURN_STRING*).
RETVAL_STRINGL(string, length, duplicate)	Sets the return value to the specified string and forces the length to become length (see

	also *RETVAL_STRING*). This macro is faster and binary-safe, and should be used whenever the string length is known.
RETVAL_EMPTY_STRING	Sets the return value to an empty string.
RETVAL_FALSE	Sets the return value to Boolean false.
RETVAL_TRUE	Sets the return value to Boolean true.

Complex types such as arrays and objects can be returned by using **array_init()** and **object_init()**, as well as the corresponding hash functions on return_value. Since these types cannot be constructed of trivial information, there are no predefined macros for them.

Printing Information

Often it's necessary to print messages to the output stream from your module, just as print() would be used within a script. PHP offers functions for most generic tasks, such as printing warning messages, generating output for phpinfo(), and so on. The following sections provide more details. Examples of these functions can be found on the CD-ROM.

zend_printf()

zend_printf() works like the standard printf(), except that it prints to Zend's output stream.

zend_error()

zend_error() can be used to generate error messages. This function accepts two arguments; the first is the error type (see *zend_errors.h*), and the second is the error message.

```
zend_error(E_WARNING, "This function has been called with empty arguments");
```

Zend's Predefined Error Messages. shows a list of possible values (see below). These values are also referred to in *php.ini*. Depending on which error type you choose, your messages will be logged.

Zend's Predefined Error Messages.

Error	Description
E_ERROR	Signals an error and terminates execution of the script immediately.
E_WARNING	Signals a generic warning. Execution continues.
E_PARSE	Signals a parser error. Execution continues.

E_NOTICE	Signals a notice. Execution continues. Note that by default the display of this type of error messages is turned off in *php.ini*.
E_CORE_ERROR	Internal error by the core; shouldn't be used by user-written modules.
E_COMPILE_ERROR	Internal error by the compiler; shouldn't be used by user-written modules.
E_COMPILE_WARNING	Internal warning by the compiler; shouldn't be used by user-written modules.

Display of warning messages in the browser.

Including Output in phpinfo()

After creating a real module, you'll want to show information about the module in phpinfo() (in addition to the module name, which appears in the module list by default). PHP allows you to create your own section in the phpinfo() output with the *ZEND_MINFO()* function. This function should be placed in the module descriptor block (discussed earlier) and is always called whenever a script calls phpinfo().

PHP automatically prints a section in phpinfo() for you if you specify the *ZEND_MINFO* function, including the module name in the heading. Everything else must be formatted and printed by you.

Typically, you can print an HTML table header using **php_info_print_table_start()** and then use the standard functions **php_info_print_table_header()** and **php_info_print_table_row()**. As arguments, both take the number of columns (as integers) and the column contents (as strings). Source code and screenshot for output in phpinfo. shows a source example and its output. To print the table footer, use **php_info_print_table_end()**.

Example #35 - Source code and screenshot for output in phpinfo().

```
php_info_print_table_start();
php_info_print_table_header(2, "First column", "Second column");
php_info_print_table_row(2, "Entry in first row", "Another entry");
php_info_print_table_row(2, "Just to fill", "another row here");
php_info_print_table_end();
```

Execution Information

You can also print execution information, such as the current file being executed. The

name of the function currently being executed can be retrieved using the function **get_active_function_name()**. This function returns a pointer to the function name and doesn't accept any arguments. To retrieve the name of the file currently being executed, use **zend_get_executed_filename()**. This function accesses the executor globals, which are passed to it using the *TSRMLS_C* macro. The executor globals are automatically available to every function that's called directly by Zend (they're part of the *INTERNAL_FUNCTION_PARAMETERS* described earlier in this chapter). If you want to access the executor globals in another function that doesn't have them available automatically, call the macro *TSRMLS_FETCH()* once in that function; this will introduce them to your local scope.

Finally, the line number currently being executed can be retrieved using the function **zend_get_executed_lineno()**. This function also requires the executor globals as arguments. For examples of these functions, see Printing execution information..

Example #36 - Printing execution information.

```
zend_printf("The name of the current function is %s&lt;br&gt;",
get_active_function_name(TSRMLS_C));
zend_printf("The file currently executed is %s&lt;br&gt;",
zend_get_executed_filename(TSRMLS_C));
zend_printf("The current line being executed is %i&lt;br&gt;",
zend_get_executed_lineno(TSRMLS_C));
```

Startup and Shutdown Functions

Startup and shutdown functions can be used for one-time initialization and deinitialization of your modules. As discussed earlier in this chapter (see the description of the Zend module descriptor block), there are module, and request startup and shutdown events.

The module startup and shutdown functions are called whenever a module is loaded and needs initialization; the request startup and shutdown functions are called every time a request is processed (meaning that a file is being executed).

For dynamic extensions, module and request startup/shutdown events happen at the same time.

Declaration and implementation of these functions can be done with macros; see the earlier section "Declaration of the Zend Module Block" for details.

Calling User Functions

You can call user functions from your own modules, which is very handy when implementing callbacks; for example, for array walking, searching, or simply for event-based programs.

User functions can be called with the function **call_user_function_ex()**. It requires a hash

value for the function table you want to access, a pointer to an object (if you want to call a method), the function name, return value, number of arguments, argument array, and a flag indicating whether you want to perform zval separation.

```
ZEND_API int call_user_function_ex(HashTable *function_table, zval *object,
zval *function_name, zval **retval_ptr_ptr,
int param_count, zval **params[],
int no_separation);
```

Note that you don't have to specify both function_table and object; either will do. If you want to call a method, you have to supply the object that contains this method, in which case **call_user_function()** automatically sets the function table to this object's function table. Otherwise, you only need to specify function_table and can set object to *NULL*.

Usually, the default function table is the "root" function table containing all function entries. This function table is part of the compiler globals and can be accessed using the macro *CG*. To introduce the compiler globals to your function, call the macro *TSRMLS_FETCH* once.

The function name is specified in a zval container. This might be a bit surprising at first, but is quite a logical step, since most of the time you'll accept function names as parameters from calling functions within your script, which in turn are contained in zval containers again. Thus, you only have to pass your arguments through to this function. This zval must be of type *IS_STRING*.

The next argument consists of a pointer to the return value. You don't have to allocate memory for this container; the function will do so by itself. However, you have to destroy this container (using **zval_dtor()**) afterward!

Next is the parameter count as integer and an array containing all necessary parameters. The last argument specifies whether the function should perform zval separation - this should always be set to *0*. If set to *1*, the function consumes less memory but fails if any of the parameters need separation.

Calling user functions. shows a small demonstration of calling a user function. The code calls a function that's supplied to it as argument and directly passes this function's return value through as its own return value. Note the use of the constructor and destructor calls at the end - it might not be necessary to do it this way here (since they should be separate values, the assignment might be safe), but this is bulletproof.

Example #37 - Calling user functions.

```
zval **function_name;
zval *retval;

if((ZEND_NUM_ARGS() != 1) || (zend_get_parameters_ex(1, &function_name) !=
SUCCESS))
{
   WRONG_PARAM_COUNT;
}

if((*function_name)->type != IS_STRING)
{
```

```
   zend_error(E_ERROR, "Function requires string argument");
}

TSRMSLS_FETCH();

if(call_user_function_ex(CG(function_table), NULL, *function_name, &retval,
0, NULL, 0) != SUCCESS)
{
   zend_error(E_ERROR, "Function call failed");
}

zend_printf("We have %i as type\n", retval->type);

*return_value = *retval;
zval_copy_ctor(return_value);
zval_ptr_dtor(&retval);
```

```
<?php

dl("call_userland.so");

function test_function()
{
   echo "We are in the test function!\n";
   return 'hello';
}

$return_value = call_userland("test_function");

echo "Return value: '$return_value'";
?>
```

The above example will output:

```
We are in the test function!
We have 3 as type
Return value: 'hello'
```

Initialization File Support

PHP 4 features a redesigned initialization file support. It's now possible to specify default initialization entries directly in your code, read and change these values at runtime, and create message handlers for change notifications.

To create an .ini section in your own module, use the macros *PHP_INI_BEGIN()* to mark the beginning of such a section and *PHP_INI_END()* to mark its end. In between you can use *PHP_INI_ENTRY()* to create entries.

```
PHP_INI_BEGIN()
PHP_INI_ENTRY("first_ini_entry",  "has_string_value", PHP_INI_ALL, NULL)
PHP_INI_ENTRY("second_ini_entry", "2",                PHP_INI_SYSTEM,
OnChangeSecond)
PHP_INI_ENTRY("third_ini_entry",  "xyz",              PHP_INI_USER, NULL)
PHP_INI_END()
```

The *PHP_INI_ENTRY()* macro accepts four parameters: the entry name, the entry value, its change permissions, and a pointer to a change-notification handler. Both entry name

and value must be specified as strings, regardless of whether they really are strings or integers.

The permissions are grouped into three sections: *PHP_INI_SYSTEM* allows a change only directly in the *php.ini* file; *PHP_INI_USER* allows a change to be overridden by a user at runtime using additional configuration files, such as *.htaccess*; and *PHP_INI_ALL* allows changes to be made without restrictions. There's also a fourth level, *PHP_INI_PERDIR*, for which we couldn't verify its behavior yet.

The fourth parameter consists of a pointer to a change-notification handler. Whenever one of these initialization entries is changed, this handler is called. Such a handler can be declared using the *PHP_INI_MH* macro:

```
PHP_INI_MH(OnChangeSecond);             // handler for ini-entry
"second_ini_entry"

// specify ini-entries here

PHP_INI_MH(OnChangeSecond)
{

   zend_printf("Message caught, our ini entry has been changed to %s&lt;br&gt;",
new_value);

   return(SUCCESS);

}
```

The new value is given to the change handler as string in the variable new_value. When looking at the definition of *PHP_INI_MH*, you actually have a few parameters to use:

```
#define PHP_INI_MH(name) int name(php_ini_entry *entry, char *new_value,
                                  uint new_value_length, void *mh_arg1,
                                  void *mh_arg2, void *mh_arg3)
```

All these definitions can be found in *php_ini.h*. Your message handler will have access to a structure that contains the full entry, the new value, its length, and three optional arguments. These optional arguments can be specified with the additional macros *PHP_INI_ENTRY1* (allowing one additional argument), *PHP_INI_ENTRY2* (allowing two additional arguments), and *PHP_INI_ENTRY3* (allowing three additional arguments).

The change-notification handlers should be used to cache initialization entries locally for faster access or to perform certain tasks that are required if a value changes. For example, if a constant connection to a certain host is required by a module and someone changes the hostname, automatically terminate the old connection and attempt a new one.

Access to initialization entries can also be handled with the macros shown in Macros to Access Initialization Entries in PHP.

Macros to Access Initialization Entries in PHP

Macro	Description
INI_INT(name)	Returns the current value of entry *name* as integer (long).
INI_FLT(name)	Returns the current value of entry *name* as

	float (double).
INI_STR(name)	Returns the current value of entry *name* as string. *Note:* This string is not duplicated, but instead points to internal data. Further access requires duplication to local memory.
INI_BOOL(name)	Returns the current value of entry *name* as Boolean (defined as zend_bool, which currently means unsigned char).
INI_ORIG_INT(name)	Returns the original value of entry *name* as integer (long).
INI_ORIG_FLT(name)	Returns the original value of entry *name* as float (double).
INI_ORIG_STR(name)	Returns the original value of entry *name* as string. Note: This string is not duplicated, but instead points to internal data. Further access requires duplication to local memory.
INI_ORIG_BOOL(name)	Returns the original value of entry *name* as Boolean (defined as zend_bool, which currently means unsigned char).

Finally, you have to introduce your initialization entries to PHP. This can be done in the module startup and shutdown functions, using the macros *REGISTER_INI_ENTRIES()* and *UNREGISTER_INI_ENTRIES()*:

```
ZEND_MINIT_FUNCTION(mymodule)
{

   REGISTER_INI_ENTRIES();

}

ZEND_MSHUTDOWN_FUNCTION(mymodule)
{

   UNREGISTER_INI_ENTRIES();

}
```

Where to Go from Here

You've learned a lot about PHP. You now know how to create dynamic loadable modules and statically linked extensions. You've learned how PHP and Zend deal with internal storage of variables and how you can create and access these variables. You know quite a set of tool functions that do a lot of routine tasks such as printing informational texts, automatically introducing variables to the symbol table, and so on.

Even though this chapter often had a mostly "referential" character, we hope that it gave you insight on how to start writing your own extensions. For the sake of space, we had to leave out a lot; we suggest that you take the time to study the header files and some modules (especially the ones in the *ext/standard* directory and the MySQL module, as these implement commonly known functionality). This will give you an idea of how other people have used the API functions - particularly those that didn't make it into this chapter.

Reference: Some Configuration Macros

config.m4

The file *config.m4* is processed by *buildconf* and must contain all the instructions to be executed during configuration. For example, these can include tests for required external files, such as header files, libraries, and so on. PHP defines a set of macros that can be used in this process, the most useful of which are described in M4 Macros for config.m4.

M4 Macros for *config.m4*

Macro	Description
AC_MSG_CHECKING(message)	Prints a "checking <message>" text during *configure*.
AC_MSG_RESULT(value)	Gives the result to *AC_MSG_CHECKING*; should specify either *yes* or *no* as value.
AC_MSG_ERROR(message)	Prints message as error message during *configure* and aborts the script.
AC_DEFINE(name,value,description)	Adds *#define* to *php_config.h* with the value of value and a comment that says description (this is useful for conditional compilation of your module).
AC_ADD_INCLUDE(path)	Adds a compiler include path; for example, used if the module needs to add search paths for header files.
AC_ADD_LIBRARY_WITH_PATH(libraryname,librarypath)	Specifies an additional library to link.
AC_ARG_WITH(modulename,description,unconditionaltest,conditionaltest)	Quite a powerful macro, adding the module with description to the *configure --help* output. PHP checks whether the option *--with-<modulename>* is given to the *configure* script. If so, it runs the script *unconditionaltest* (for example, *--with-myext=yes*), in which case the value of the option is contained in the variable

	$withval. Otherwise, it executes *conditionaltest*.
PHP_EXTENSION(modulename, [shared])	This macro is a *must* to call for PHP to configure your extension. You can supply a second argument in addition to your module name, indicating whether you intend compilation as a shared module. This will result in a definition at compile time for your source as *COMPILE_DL_<modulename>*.

API Macros

A set of macros was introduced into Zend's API that simplify access to zval containers (see API Macros for Accessing zval Containers).

API Macros for Accessing zval Containers

Macro	Refers to
Z_LVAL(zval)	(zval).value.lval
Z_DVAL(zval)	(zval).value.dval
Z_STRVAL(zval)	(zval).value.str.val
Z_STRLEN(zval)	(zval).value.str.len
Z_ARRVAL(zval)	(zval).value.ht
Z_LVAL_P(zval)	(*zval).value.lval
Z_DVAL_P(zval)	(*zval).value.dval
Z_STRVAL_P(zval_p)	(*zval).value.str.val
Z_STRLEN_P(zval_p)	(*zval).value.str.len
Z_ARRVAL_P(zval_p)	(*zval).value.ht
Z_LVAL_PP(zval_pp)	(**zval).value.lval
Z_DVAL_PP(zval_pp)	(**zval).value.dval
Z_STRVAL_PP(zval_pp)	(**zval).value.str.val
Z_STRLEN_PP(zval_pp)	(**zval).value.str.len
Z_ARRVAL_PP(zval_pp)	(**zval).value.ht

TSRM API

The future: PHP 6 and Zend Engine 3