Error Handling and Logging

Introduction

These are functions dealing with error handling and logging. They allow you to define your own error handling rules, as well as modify the way the errors can be logged. This allows you to change and enhance error reporting to suit your needs.

With the logging functions, you can send messages directly to other machines, to an email (or email to pager gateway!), to system logs, etc., so you can selectively log and monitor the most important parts of your applications and websites.

The error reporting functions allow you to customize what level and kind of error feedback is given, ranging from simple notices to customized functions returned during errors.

Installing/Configuring

Requirements

No external libraries are needed to build this extension.

Installation

There is no installation needed to use these functions; they are part of the PHP core.

Runtime Configuration

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

Errors and Logging Configuration Options

Name	Default	Changeable	Changelog
error_reporting	NULL	PHP_INI_ALL	
display_errors	"1"	PHP_INI_ALL	
display_startup_errors	"0"	PHP_INI_ALL	
log_errors	"0"	PHP_INI_ALL	
log_errors_max_len	"1024"	PHP_INI_ALL	Available since PHP 4.3.0.
ignore_repeated_errors	"0"	PHP_INI_ALL	Available since PHP 4.3.0.
ignore_repeated_source	"0"	PHP_INI_ALL	Available since PHP 4.3.0.
report_memleaks	"1"	PHP_INI_ALL	Available since PHP 4.3.0.
track_errors	"0"	PHP_INI_ALL	
html_errors	"1"	PHP_INI_ALL	PHP_INI_SYSTEM in PHP <= 4.2.3.

docref_root	""	PHP_INI_ALL	Available since PHP 4.3.0.
docref_ext	""	PHP_INI_ALL	Available since PHP 4.3.2.
error_prepend_string	NULL	PHP_INI_ALL	
error_append_string	NULL	PHP_INI_ALL	
error_log	NULL	PHP_INI_ALL	

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

Here's a short explanation of the configuration directives.

error_reporting integer
Set the error reporting level. The parameter is either an integer representing a bit field, or named constants. The error_reporting levels and constants are described in Predefined Constants, and in *php.ini*. To set at runtime, use the error_reporting() function. See also the display_errors directive. In PHP 4 and PHP 5 the default value is E_ALL & ~E_NOTICE. This setting does not show **E_NOTICE** level errors. You may want to show them during development.

Note

Enabling **E_NOTICE** during development has some benefits. For debugging purposes: NOTICE messages will warn you about possible bugs in your code. For example, use of unassigned values is warned. It is extremely useful to find typos and to save time for debugging. NOTICE messages will warn you about bad style. For example, $arr[item] is better to be written as $arr['item'] since PHP tries to treat "item" as constant. If it is not a constant, PHP assumes it is a string index for the array.

Note

In PHP 5 a new error level **E_STRICT** is available. As **E_STRICT** is not included within **E_ALL** you have to explicitly enable this kind of error level. Enabling **E_STRICT** during development has some benefits. STRICT messages will help you to use the latest and greatest suggested method of coding, for example warn you about using deprecated functions.

Note

PHP Constants outside of PHP

Using PHP Constants outside of PHP, like in *httpd.conf*, will have no useful

meaning so in such cases the integer values are required. And since error levels will be added over time, the maximum value (for **E_ALL**) will likely change. So in place of **E_ALL** consider using a larger value to cover all bit fields from now and well into the future, a numeric value like *2147483647*.

display_errors string
This determines whether errors should be printed to the screen as part of the output or if they should be hidden from the user. Value "stderr" sends the errors to *stderr* instead of *stdout*. The value is available as of PHP 5.2.4. In earlier versions, this directive was of type boolean.

Note

This is a feature to support your development and should never be used on production systems (e.g. systems connected to the internet).

Note

Although display_errors may be set at runtime (with ini_set()), it won't have any affect if the script has fatal errors. This is because the desired runtime action does not get executed.

display_startup_errors boolean
Even when display_errors is on, errors that occur during PHP's startup sequence are not displayed. It's strongly recommended to keep display_startup_errors off, except for debugging.

log_errors boolean
Tells whether script error messages should be logged to the server's error log or error_log. This option is thus server-specific.

Note

You're strongly advised to use error logging in place of error displaying on production web sites.

log_errors_max_len integer
Set the maximum length of log_errors in bytes. In error_log information about the source is added. The default is 1024 and 0 allows to not apply any maximum length at all. This length is applied to logged errors, displayed errors and also to $php_errormsg. When an integer is used, the value is measured in bytes. Shorthand notation, as described in this FAQ, may also be used.

ignore_repeated_errors boolean

Do not log repeated messages. Repeated errors must occur in the same file on the same line until ignore_repeated_source is set true.

ignore_repeated_source boolean
Ignore source of message when ignoring repeated messages. When this setting is On you will not log errors with repeated messages from different files or sourcelines.

report_memleaks boolean
If this parameter is set to Off, then memory leaks will not be shown (on stdout or in the log). This has only effect in a debug compile, and if error_reporting includes E_WARNING in the allowed list

track_errors boolean
If enabled, the last error message will always be present in the variable $php_errormsg .

html_errors boolean
Turn off HTML tags in error messages. The new format for HTML errors produces clickable messages that direct the user to a page describing the error or function in causing the error. These references are affected by docref_root and docref_ext.

docref_root string
The new error format contains a reference to a page describing the error or function causing the error. In case of manual pages you can download the manual in your language and set this ini directive to the URL of your local copy. If your local copy of the manual can be reached by '/manual/' you can simply use **docref_root=/manual/**. Additional you have to set docref_ext to match the fileextensions of your copy **docref_ext=.html**. It is possible to use external references. For example you can use **docref_root=http://manual/en/** or **docref_root="http://landonize.it/?how=url&theme=classic&filter=Landon &url=http%3A%2F%2Fwww.php.net%2F"** Most of the time you want the docref_root value to end with a slash '/'. But see the second example above which does not have nor need it.

Note
This is a feature to support your development since it makes it easy to lookup a function description. However it should never be used on production systems (e.g. systems connected to the internet).

docref_ext string
See docref_root.

Note
The value of docref_ext must begin with a dot '.'.

error_prepend_string string

String to output before an error message.

error_append_string string
String to output after an error message.

error_log string
Name of the file where script errors should be logged. The file should be writable by the web server's user. If the special value *syslog* is used, the errors are sent to the system logger instead. On Unix, this means syslog(3) and on Windows NT it means the event log. The system logger is not supported on Windows 95. See also: syslog(). If this directive is not set, errors are sent to the SAPI error logger. For example, it is an error log in Apache or *stderr* in CLI.

Resource Types

This extension has no resource types defined.

Predefined Constants

The constants below are always available as part of the PHP core.

Note
You may use these constant names in *php.ini* but not outside of PHP, like in *httpd.conf*, where you'd use the bitmask values instead.

Errors and Logging

Value	Constant	Description	Note
1	**E_ERROR** (integer)	Fatal run-time errors. These indicate errors that can not be recovered from, such as a memory allocation problem. Execution of the script is halted.	
2	**E_WARNING** (integer)	Run-time warnings (non-fatal errors). Execution of the script is not halted.	
4	**E_PARSE** (integer)	Compile-time parse errors. Parse errors should only be generated by the parser.	
8	**E_NOTICE** (integer)	Run-time notices. Indicate that the script encountered something that could indicate an error, but could also happen in the normal course of running a script.	
16	**E_CORE_ERROR** (integer)	Fatal errors that occur during PHP's initial startup. This is like an **E_ERROR**, except it is generated	since PHP 4

		by the core of PHP.	
32	**E_CORE_WARNING** (integer)	Warnings (non-fatal errors) that occur during PHP's initial startup. This is like an **E_WARNING**, except it is generated by the core of PHP.	since PHP 4
64	**E_COMPILE_ERROR** (integer)	Fatal compile-time errors. This is like an **E_ERROR**, except it is generated by the Zend Scripting Engine.	since PHP 4
128	**E_COMPILE_WARNING** (integer)	Compile-time warnings (non-fatal errors). This is like an **E_WARNING**, except it is generated by the Zend Scripting Engine.	since PHP 4
256	**E_USER_ERROR** (integer)	User-generated error message. This is like an **E_ERROR**, except it is generated in PHP code by using the PHP function trigger_error().	since PHP 4
512	**E_USER_WARNING** (integer)	User-generated warning message. This is like an **E_WARNING**, except it is generated in PHP code by using the PHP function trigger_error().	since PHP 4
1024	**E_USER_NOTICE** (integer)	User-generated notice message. This is like an **E_NOTICE**, except it is generated in PHP code by using the PHP function trigger_error().	since PHP 4
2048	**E_STRICT** (integer)	Run-time notices. Enable to have PHP	since PHP 5

		suggest changes to your code which will ensure the best interoperability and forward compatibility of your code.	
4096	**E_RECOVERABLE_ERROR** (integer)	Catchable fatal error. It indicates that a probably dangerous error occured, but did not leave the Engine in an unstable state. If the error is not caught by a user defined handle (see also set_error_handler()), the application aborts as it was an **E_ERROR**.	since PHP 5.2.0
8191	**E_ALL** (integer)	All errors and warnings, as supported, except of level **E_STRICT** in PHP < 6.	6143 in PHP 5.2.x and 2047 previously

The above values (either numerical or symbolic) are used to build up a bitmask that specifies which errors to report. You can use the bitwise operators to combine these values or mask out certain types of errors. Note that only '|', '~', '!', '^' and '&' will be understood within *php.ini*.

Examples

Below we can see an example of using the error handling capabilities in PHP. We define an error handling function which logs the information into a file (using an XML format), and e-mails the developer in case a critical error in the logic happens.

Example #1 - Using error handling in a script

```
<?php
// we will do our own error handling
error_reporting(0);

// user defined error handling function
function userErrorHandler($errno, $errmsg, $filename, $linenum, $vars)
{
   // timestamp for the error entry
   $dt = date("Y-m-d H:i:s (T)");

   // define an assoc array of error string
   // in reality the only entries we should
   // consider are E_WARNING, E_NOTICE, E_USER_ERROR,
   // E_USER_WARNING and E_USER_NOTICE
   $errortype = array (
               E_ERROR              => 'Error',
               E_WARNING            => 'Warning',
               E_PARSE              => 'Parsing Error',
               E_NOTICE             => 'Notice',
               E_CORE_ERROR         => 'Core Error',
               E_CORE_WARNING       => 'Core Warning',
               E_COMPILE_ERROR      => 'Compile Error',
               E_COMPILE_WARNING    => 'Compile Warning',
               E_USER_ERROR         => 'User Error',
               E_USER_WARNING       => 'User Warning',
               E_USER_NOTICE        => 'User Notice',
               E_STRICT             => 'Runtime Notice',
               E_RECOVERABLE_ERROR  => 'Catchable Fatal Error'
               );
   // set of errors for which a var trace will be saved
   $user_errors = array(E_USER_ERROR, E_USER_WARNING, E_USER_NOTICE);

   $err = "<errorentry>\n";
   $err .= "\t<datetime>" . $dt . "</datetime>\n";
   $err .= "\t<errornum>" . $errno . "</errornum>\n";
   $err .= "\t<errortype>" . $errortype[$errno] . "</errortype>\n";
   $err .= "\t<errormsg>" . $errmsg . "</errormsg>\n";
   $err .= "\t<scriptname>" . $filename . "</scriptname>\n";
   $err .= "\t<scriptlinenum>" . $linenum . "</scriptlinenum>\n";

   if (in_array($errno, $user_errors)) {
       $err .= "\t<vartrace>" . wddx_serialize_value($vars, "Variables") .
"</vartrace>\n";
   }
   $err .= "</errorentry>\n\n";

   // for testing
   // echo $err;
```

```
    // save to the error log, and e-mail me if there is a critical user error
    error_log($err, 3, "/usr/local/php4/error.log");
    if ($errno == E_USER_ERROR) {
        mail("phpdev@example.com", "Critical User Error", $err);
    }
}


function distance($vect1, $vect2)
{
    if (!is_array($vect1) || !is_array($vect2)) {
        trigger_error("Incorrect parameters, arrays expected", E_USER_ERROR);
        return NULL;
    }

    if (count($vect1) != count($vect2)) {
        trigger_error("Vectors need to be of the same size", E_USER_ERROR);
        return NULL;
    }

    for ($i=0; $i<count($vect1); $i++) {
        $c1 = $vect1[$i]; $c2 = $vect2[$i];
        $d = 0.0;
        if (!is_numeric($c1)) {
            trigger_error("Coordinate $i in vector 1 is not a number, using
zero",
                          E_USER_WARNING);
            $c1 = 0.0;
        }
        if (!is_numeric($c2)) {
            trigger_error("Coordinate $i in vector 2 is not a number, using
zero",
                          E_USER_WARNING);
            $c2 = 0.0;
        }
        $d += $c2*$c2 - $c1*$c1;
    }
    return sqrt($d);
}

$old_error_handler = set_error_handler("userErrorHandler");

// undefined constant, generates a warning
$t = I_AM_NOT_DEFINED;

// define some "vectors"
$a = array(2, 3, "foo");
$b = array(5.5, 4.3, -1.6);
$c = array(1, -3);

// generate a user error
$t1 = distance($c, $b) . "\n";

// generate another user error
$t2 = distance($b, "i am not an array") . "\n";

// generate a warning
$t3 = distance($a, $b) . "\n";

?>
```

Error Handling Functions

See Also

See also syslog().

debug_backtrace

debug_backtrace -- Generates a backtrace

Description

array **debug_backtrace** ([bool `$provide_object`])

debug_backtrace() generates a PHP backtrace.

Return Values

Returns an associative array. The possible returned elements are as follows:

Possible returned elements from debug_backtrace()

Name	Type	Description
function	string	The current function name. See also __FUNCTION__.
line	integer	The current line number. See also __LINE__.
file	string	The current file name. See also __FILE__.
class	string	The current class name. See also __CLASS__
object	object	The current object.
type	string	The current call type. If a method call, "->" is returned. If a static method call, "::" is returned. If a function call, nothing is returned.
args	array	If inside a function, this lists the functions arguments. If inside an included file, this lists the included file name(s).

ChangeLog

Version	Description
5.2.5	Added the optional parameter *provide_object*.
5.1.1	Added the current object as a possible return element.

Examples

Example #2 - debug_backtrace() example

```
<?php
// filename: a.php

function a_test($str)
{
   echo "\nHi: $str";
   var_dump(debug_backtrace());
}

a_test('friend');
?>

<?php
// filename: b.php
include_once '/tmp/a.php';
?>
```

Results similar to the following when executing */tmp/b.php*:

```
Hi: friend
array(2) {
[0]=>
array(4) {
   ["file"] => string(10) "/tmp/a.php"
   ["line"] => int(10)
   ["function"] => string(6) "a_test"
   ["args"]=>
   array(1) {
     [0] => &string(6) "friend"
   }
}
[1]=>
array(4) {
   ["file"] => string(10) "/tmp/b.php"
   ["line"] => int(2)
   ["args"] =>
   array(1) {
     [0] => string(10) "/tmp/a.php"
   }
   ["function"] => string(12) "include_once"
 }
```

```
}
```

See Also

- trigger_error()
- debug_print_backtrace()

debug_print_backtrace

debug_print_backtrace -- Prints a backtrace

Description

void **debug_print_backtrace** (void)

debug_print_backtrace() prints a PHP backtrace. It prints the function calls, included/required files and eval() ed stuff.

Parameters

This function has no parameters.

Return Values

No value is returned.

Examples

Example #3 - debug_print_backtrace() example

```
<?php
// include.php file

function a() {
   b();
}

function b() {
   c();
}

function c(){
   debug_print_backtrace();
}

a();

?>

<?php
// test.php file
// this is the file you should run

include 'include.php';
?>
```

The above example will output something similar to:

```
#0  eval() called at [/tmp/include.php:5]
#1  a() called at [/tmp/include.php:17]
#2  include(/tmp/include.php) called at [/tmp/test.php:3]

#0  c() called at [/tmp/include.php:10]
#1  b() called at [/tmp/include.php:6]
#2  a() called at [/tmp/include.php:17]
#3  include(/tmp/include.php) called at [/tmp/test.php:3]
```

See Also

- debug_backtrace()

error_get_last

error_get_last -- Get the last occurred error

Description

array **error_get_last** (void)

Gets information about the last error that occured.

Return Values

Returns an associative array describing the last error with keys "type", "message", "file" and "line". Returns **NULL** if there hasn't been an error yet.

Examples

Example #4 - An error_get_last() example

```
<?php
echo $a;
print_r(error_get_last());
?>
```

The above example will output something similar to:

```
Array
(
    [type] => 8
    [message] => Undefined variable: a
    [file] => C:\WWW\index.php
    [line] => 2
)
```

See Also

- Error constants
- Variable *$php_errormsg*
- Directive *display_errors*

error_log

error_log -- Send an error message somewhere

Description

bool **error_log** (string `$message` [, int `$message_type` [, string `$destination` [, string `$extra_headers`]]])

Sends an error message to the web server's error log, a TCP port or to a file.

Parameters

`message`
The error message that should be logged.

`message_type`
Says where the error should go. The possible message types are as follows:

error_log() **log types**

0	*`message`* is sent to PHP's system logger, using the Operating System's system logging mechanism or a file, depending on what the error_log configuration directive is set to. This is the default option.
1	*`message`* is sent by email to the address in the *`destination`* parameter. This is the only message type where the fourth parameter, *`extra_headers`* is used.
2	No longer an option.
3	*`message`* is appended to the file *`destination`* . A newline is not automatically added to the end of the *`message`* string.

`destination`
The destination. Its meaning depends on the *`message_type`* parameter as described above.

`extra_headers`
The extra headers. It's used when the *`message_type`* parameter is set to *1*. This message type uses the same internal function as mail() does.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

Examples

Example #5 - error_log() examples

```
<?php
// Send notification through the server log if we can not
// connect to the database.
if (!Ora_Logon($username, $password)) {
   error_log("Oracle database not available!", 0);
}

// Notify administrator by email if we run out of FOO
if (!($foo = allocate_new_foo())) {
   error_log("Big trouble, we're all out of FOOs!", 1,
             "operator@example.com");
}

// another way to call error_log():
error_log("You messed up!", 3, "/var/tmp/my-errors.log");
?>
```

error_reporting

error_reporting -- Sets which PHP errors are reported

Description

int **error_reporting** ([int `$level`])

The error_reporting() function sets the error_reporting directive at runtime. PHP has many levels of errors, using this function sets that level for the duration (runtime) of your script.

Parameters

`level`

The new error_reporting level. It takes on either a bitmask, or named constants. Using named constants is strongly encouraged to ensure compatibility for future versions. As error levels are added, the range of integers increases, so older integer-based error levels will not always behave as expected. The available error level constants are listed below. The actual meanings of these error levels are described in the predefined constants.

error_reporting() **level constants and bit values**

value	constant
1	E_ERROR
2	E_WARNING
4	E_PARSE
8	E_NOTICE
16	E_CORE_ERROR
32	E_CORE_WARNING
64	E_COMPILE_ERROR
128	E_COMPILE_WARNING
256	E_USER_ERROR
512	E_USER_WARNING
1024	E_USER_NOTICE
6143	E_ALL

2048	E_STRICT
4096	E_RECOVERABLE_ERROR

Return Values

Returns the old error_reporting level.

ChangeLog

Version	Description
5.0.0	**E_STRICT** introduced (not part of **E_ALL**).
5.2.0	**E_RECOVERABLE_ERROR** introduced.
6	**E_STRICT** became part of **E_ALL**.

Examples

Example #6 - error_reporting() examples

```
<?php

// Turn off all error reporting
error_reporting(0);

// Report simple running errors
error_reporting(E_ERROR | E_WARNING | E_PARSE);

// Reporting E_NOTICE can be good too (to report uninitialized
// variables or catch variable name misspellings ...)
error_reporting(E_ERROR | E_WARNING | E_PARSE | E_NOTICE);

// Report all errors except E_NOTICE
// This is the default value set in php.ini
error_reporting(E_ALL ^ E_NOTICE);

// Report all PHP errors
error_reporting(E_ALL);

// Same as error_reporting(E_ALL);
ini_set('error_reporting', E_ALL);

?>
```

Notes

Warning
Most of **E_STRICT** errors are evaluated at the compile time thus such errors are not reported in the file where error_reporting is enhanced to include **E_STRICT** errors (and vice versa).

See Also

- The display_errors directive
- ini_set()

restore_error_handler

restore_error_handler -- Restores the previous error handler function

Description

bool **restore_error_handler** (void)

Used after changing the error handler function using set_error_handler(), to revert to the previous error handler (which could be the built-in or a user defined function).

Return Values

This function always returns **TRUE**.

Examples

Example #7 - restore_error_handler() example

Decide if unserialize() caused an error, then restore the original error handler.

```
<?php
function unserialize_handler($errno, $errstr)
{
   echo "Invalid serialized value.\n";
}

$serialized = 'foo';
set_error_handler('unserialize_handler');
$original = unserialize($serialized);
restore_error_handler();
?>
```

The above example will output:

```
Invalid serialized value.
```

Notes

Note

Calling restore_error_handler() from the *error_handler* function is ignored.

See Also

- error_reporting()
- set_error_handler()
- restore_exception_handler()
- trigger_error()

restore_exception_handler

restore_exception_handler -- Restores the previously defined exception handler function

Description

bool **restore_exception_handler** (void)

Used after changing the exception handler function using set_exception_handler(), to revert to the previous exception handler (which could be the built-in or a user defined function).

Return Values

This function always returns **TRUE**.

See Also

- set_exception_handler()
- set_error_handler()
- restore_error_handler()
- error_reporting()

set_error_handler

set_error_handler -- Sets a user-defined error handler function

Description

mixed **set_error_handler** (callback `$error_handler` [, int `$error_types`])

Sets a user function (*`error_handler`*) to handle errors in a script.

This function can be used for defining your own way of handling errors during runtime, for example in applications in which you need to do cleanup of data/files when a critical error happens, or when you need to trigger an error under certain conditions (using trigger_error()).

It is important to remember that the standard PHP error handler is completely bypassed. error_reporting() settings will have no effect and your error handler will be called regardless - however you are still able to read the current value of error_reporting and act appropriately. Of particular note is that this value will be 0 if the statement that caused the error was prepended by the @ error-control operator.

Also note that it is your responsibility to die() if necessary. If the error-handler function returns, script execution will continue with the next statement after the one that caused an error.

The following error types cannot be handled with a user defined function: **E_ERROR**, **E_PARSE**, **E_CORE_ERROR**, **E_CORE_WARNING**, **E_COMPILE_ERROR**, **E_COMPILE_WARNING**, and most of **E_STRICT** raised in the file where set_error_handler() is called.

If errors occur before the script is executed (e.g. on file uploads) the custom error handler cannot be called since it is not registered at that time.

Parameters

`error_handler`

The user function needs to accept two parameters: the error code, and a string describing the error. Then there are three optional parameters that may be supplied: the filename in which the error occurred, the line number in which the error occurred, and the context in which the error occurred (an array that points to the active symbol table at the point the error occurred). The function can be shown as:
handler (int `$errno`, string `$errstr` [, string `$errfile` [, int `$errline` [, array `$errcontext`]]])

`errno`

The first parameter, *`errno`*, contains the level of the error raised, as an integer.

`errstr`

The second parameter, *errstr*, contains the error message, as a string.

errfile
The third parameter is optional, *errfile*, which contains the filename that the error was raised in, as a string.

errline
The fourth parameter is optional, *errline*, which contains the line number the error was raised at, as an integer.

errcontext
The fifth parameter is optional, *errcontext*, which is an array that points to the active symbol table at the point the error occurred. In other words, *errcontext* will contain an array of every variable that existed in the scope the error was triggered in. User error handler must not modify error context.

If the function returns **FALSE** then the normal error handler continues.

error_types
Can be used to mask the triggering of the *error_handler* function just like the error_reporting ini setting controls which errors are shown. Without this mask set the *error_handler* will be called for every error regardless to the setting of the error_reporting setting.

Return Values

Returns a string containing the previously defined error handler (if any), or **NULL** on error. If the previous handler was a class method, this function will return an indexed array with the class and the method name.

ChangeLog

Version	Description
5.2.0	The error handler must return **FALSE** to populate $php_errormsg.
5.0.0	The *error_types* parameter was introduced.
4.3.0	Instead of a function name, an array containing an object reference and a method name can also be supplied as the *error_handler*.
4.0.2	Three optional parameters for the *error_handler* user function was introduced. These are the filename, the line number, and the context.

Examples

Example #8 - Error handling with set_error_handler() and trigger_error()

The example below shows the handling of internal exceptions by triggering errors and handling them with a user defined function:

```
<?php
// error handler function
function myErrorHandler($errno, $errstr, $errfile, $errline)
{
   switch ($errno) {
   case E_USER_ERROR:
       echo "<b>My ERROR</b> [$errno] $errstr<br />\n";
       echo "  Fatal error on line $errline in file $errfile";
       echo ", PHP " . PHP_VERSION . " (" . PHP_OS . ")<br />\n";
       echo "Aborting...<br />\n";
       exit(1);
       break;

   case E_USER_WARNING:
       echo "<b>My WARNING</b> [$errno] $errstr<br />\n";
       break;

   case E_USER_NOTICE:
       echo "<b>My NOTICE</b> [$errno] $errstr<br />\n";
       break;

   default:
       echo "Unknown error type: [$errno] $errstr<br />\n";
       break;
   }

   /* Don't execute PHP internal error handler */
   return true;
}

// function to test the error handling
function scale_by_log($vect, $scale)
{
   if (!is_numeric($scale) || $scale <= 0) {
       trigger_error("log(x) for x <= 0 is undefined, you used: scale =
$scale", E_USER_ERROR);
   }

   if (!is_array($vect)) {
       trigger_error("Incorrect input vector, array of values expected",
E_USER_WARNING);
       return null;
   }

   $temp = array();
   foreach($vect as $pos => $value) {
       if (!is_numeric($value)) {
           trigger_error("Value at position $pos is not a number, using 0
(zero)", E_USER_NOTICE);
           $value = 0;
```

```
        }
        $temp[$pos] = log($scale) * $value;
    }

    return $temp;
}

// set to the user defined error handler
$old_error_handler = set_error_handler("myErrorHandler");

// trigger some errors, first define a mixed array with a non-numeric item
echo "vector a\n";
$a = array(2, 3, "foo", 5.5, 43.3, 21.11);
print_r($a);

// now generate second array
echo "----\nvector b - a notice (b = log(PI) * a)\n";
/* Value at position $pos is not a number, using 0 (zero) */
$b = scale_by_log($a, M_PI);
print_r($b);

// this is trouble, we pass a string instead of an array
echo "----\nvector c - a warning\n";
/* Incorrect input vector, array of values expected */
$c = scale_by_log("not array", 2.3);
var_dump($c); // NULL

// this is a critical error, log of zero or negative number is undefined
echo "----\nvector d - fatal error\n";
/* log(x) for x <= 0 is undefined, you used: scale = $scale" */
$d = scale_by_log($a, -2.5);
var_dump($d); // Never reached
?>
```

The above example will output something similar to:

```
vector a
Array
(
    [0] => 2
    [1] => 3
    [2] => foo
    [3] => 5.5
    [4] => 43.3
    [5] => 21.11
)
----
vector b - a notice (b = log(PI) * a)
<b>My NOTICE</b> [1024] Value at position 2 is not a number, using 0
(zero)<br />
Array
(
    [0] => 2.2894597716988
    [1] => 3.4341896575482
    [2] => 0
    [3] => 6.2960143721717
    [4] => 49.566804057279
    [5] => 24.165247890281
)
----
```

```
vector c - a warning
<b>My WARNING</b> [512] Incorrect input vector, array of values expected<br
/>
NULL
----
vector d - fatal error
<b>My ERROR</b> [256] log(x) for x <= 0 is undefined, you used: scale =
-2.5<br />
 Fatal error on line 35 in file trigger_error.php, PHP 5.2.1 (FreeBSD)<br />
Aborting...<br />
```

See Also

- error_reporting()
- restore_error_handler()
- trigger_error()
- error level constants
- information about the callback type

set_exception_handler

set_exception_handler -- Sets a user-defined exception handler function

Description

string **set_exception_handler** (callback `$exception_handler`)

Sets the default exception handler if an exception is not caught within a try/catch block. Execution will stop after the *`exception_handler`* is called.

Parameters

`exception_handler`
: Name of the function to be called when an uncaught exception occurs. This function must be defined before calling set_exception_handler(). This handler function needs to accept one parameter, which will be the exception object that was thrown.

Return Values

Returns the name of the previously defined exception handler, or **NULL** on error. If no previous handler was defined, **NULL** is also returned.

Examples

Example #9 - set_exception_handler() example

```
<?php
function exception_handler($exception) {
 echo "Uncaught exception: " , $exception->getMessage(), "\n";
}

set_exception_handler('exception_handler');

throw new Exception('Uncaught Exception');
echo "Not Executed\n";
?>
```

See Also

restore_exception_handler(), restore_error_handler(), error_reporting(), information about the callback type, and PHP 5 Exceptions.

trigger_error

trigger_error -- Generates a user-level error/warning/notice message

Description

bool **trigger_error** (string `$error_msg` [, int `$error_type`])

Used to trigger a user error condition, it can be used by in conjunction with the built-in error handler, or with a user defined function that has been set as the new error handler (set_error_handler()).

This function is useful when you need to generate a particular response to an exception at runtime.

Parameters

`error_msg`
The designated error message for this error. It's limited to 1024 characters in length. Any additional characters beyond 1024 will be truncated.

`error_type`
The designated error type for this error. It only works with the E_USER family of constants, and will default to **E_USER_NOTICE**.

Return Values

This function returns **FALSE** if wrong *`error_type`* is specified, **TRUE** otherwise.

Examples

Example #10 - trigger_error() example

See set_error_handler() for a more extensive example.

```
<?php
if (assert($divisor == 0)) {
   trigger_error("Cannot divide by zero", E_USER_ERROR);
}
?>
```

See Also

- error_reporting()
- set_error_handler()
- restore_error_handler()
- The error level constants

user_error

user_error -- Alias of trigger_error()

Description

This function is an alias of: trigger_error().