Streams

Introduction

Streams were introduced with PHP 4.3.0 as a way of generalizing file, network, data compression, and other operations which share a common set of functions and uses. In its simplest definition, a *stream* is a *resource* object which exhibits streamable behavior. That is, it can be read from or written to in a linear fashion, and may be able to fseek() to an arbitrary locations within the stream.

A *wrapper* is additional code which tells the stream how to handle specific protocols/encodings. For example, the *http* wrapper knows how to translate a URL into an *HTTP/1.0* request for a file on a remote server. There are many wrappers built into PHP by default (See List of Supported Protocols/Wrappers), and additional, custom wrappers may be added either within a PHP script using stream_wrapper_register(), or directly from an extension using the API Reference in Working with streams. Because any variety of wrapper may be added to PHP, there is no set limit on what can be done with them. To access the list of currently registered wrappers, use stream_get_wrappers().

A stream is referenced as: `scheme`:// `target`

- `scheme` (string) - The name of the wrapper to be used. Examples include: file, http, https, ftp, ftps, compress.zlib, compress.bz2, and php. See List of Supported Protocols/Wrappers for a list of PHP built-in wrappers. If no wrapper is specified, the function default is used (typically *file*://).
- `target` - Depends on the wrapper used. For filesystem related streams this is typically a path and filename of the desired file. For network related streams this is typically a hostname, often with a path appended. Again, see List of Supported Protocols/Wrappers for a description of targets for built-in streams.

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

Installing/Configuring

Requirements

No external libraries are needed to build this extension.

Installation

Streams are an integral part of PHP as of version 4.3.0. No steps are required to enable them.

Runtime Configuration

This extension has no configuration directives defined in *php.ini*.

Stream Classes

User designed wrappers can be registered via stream_wrapper_register(), using the class definition shown on that manual page.

class php_user_filter is predefined and is an abstract baseclass for use with user defined filters. See the manual page for stream_filter_register() for details on implementing user defined filters.

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.

Constant	Description
STREAM_FILTER_READ *	Used with stream_filter_append() and stream_filter_prepend() to indicate that the specified filter should only be applied when *reading*
STREAM_FILTER_WRITE *	Used with stream_filter_append() and stream_filter_prepend() to indicate that the specified filter should only be applied when *writing*
STREAM_FILTER_ALL *	This constant is equivalent to `STREAM_FILTER_READ | STREAM_FILTER_WRITE`
PSFS_PASS_ON *	*Return Code* indicating that the userspace filter returned buckets in `$out`.
PSFS_FEED_ME *	*Return Code* indicating that the userspace filter did not return buckets in `$out` (i.e. No data available).
PSFS_ERR_FATAL *	*Return Code* indicating that the userspace filter encountered an unrecoverable error (i.e. Invalid data received).
STREAM_USE_PATH	*Flag* indicating if the *stream* used the include path.
STREAM_REPORT_ERRORS	*Flag* indicating if the *wrapper* is responsible for raising errors using trigger_error() during opening of the stream. If this flag is not set, you should not raise any errors.
STREAM_CLIENT_ASYNC_CONNECT *	Open client socket asynchronously. This option must be used together with the **STREAM_CLIENT_CONNECT** flag. Used with stream_socket_client().
STREAM_CLIENT_CONNECT *	Open client socket connection. Client sockets should always include this flag. Used with stream_socket_client().

STREAM_CLIENT_PERSISTENT *	Client socket opened with stream_socket_client() should remain persistent between page loads.
STREAM_SERVER_BIND *	Tells a stream created with stream_socket_server() to bind to the specified target. Server sockets should always include this flag.
STREAM_SERVER_LISTEN *	Tells a stream created with stream_socket_server() and bound using the **STREAM_SERVER_BIND** flag to start listening on the socket. Connection-orientated transports (such as TCP) must use this flag, otherwise the server socket will not be enabled. Using this flag for connect-less transports (such as UDP) is an error.
STREAM_NOTIFY_RESOLVE *	A remote address required for this stream has been resolved, or the resolution failed. See *severity* for an indication of which happened.
STREAM_NOTIFY_CONNECT	A connection with an external resource has been established.
STREAM_NOTIFY_AUTH_REQUIRED	Additional authorization is required to access the specified resource. Typical issued with *severity* level of **STREAM_NOTIFY_SEVERITY_ERR**.
STREAM_NOTIFY_MIME_TYPE_IS	The *mime-type* of resource has been identified, refer to *message* for a description of the discovered type.
STREAM_NOTIFY_FILE_SIZE_IS	The *size* of the resource has been discovered.
STREAM_NOTIFY_REDIRECTED	The external resource has redirected the stream to an alternate location. Refer to *message*.
STREAM_NOTIFY_PROGRESS	Indicates current progress of the stream transfer in *bytes_transferred* and possibly *bytes_max* as well.
STREAM_NOTIFY_COMPLETED *	There is no more data available on the stream.
STREAM_NOTIFY_FAILURE	A generic error occurred on the stream, consult *message* and *message_code* for

	details.
STREAM_NOTIFY_AUTH_RESULT	Authorization has been completed (with or without success).
STREAM_NOTIFY_SEVERITY_INFO	Normal, non-error related, notification.
STREAM_NOTIFY_SEVERITY_WARN	Non critical error condition. Processing may continue.
STREAM_NOTIFY_SEVERITY_ERR	A critical error occurred. Processing cannot continue.
STREAM_IPPROTO_ICMP +	Provides a ICMP socket.
STREAM_IPPROTO_IP +	Provides a IP socket.
STREAM_IPPROTO_RAW +	Provides a RAW socket.
STREAM_IPPROTO_TCP +	Provides a TCP socket.
STREAM_IPPROTO_UDP +	Provides a UDP socket.
STREAM_PF_INET +	Internet Protocol Version 4 (IPv4).
STREAM_PF_INET6 +	Internet Protocol Version 6 (IPv6).
STREAM_PF_UNIX +	Unix system internal protocols.
STREAM_SOCK_DGRAM +	Provides datagrams, which are connectionless messages (UDP, for example).
STREAM_SOCK_RAW +	Provides a raw socket, which provides access to internal network protocols and interfaces. Usually this type of socket is just available to the root user.
STREAM_SOCK_RDM +	Provides a RDM (Reliably-delivered messages) socket.
STREAM_SOCK_SEQPACKET +	Provides a sequenced packet stream socket.
STREAM_SOCK_STREAM +	Provides sequenced, two-way byte streams with a transmission mechanism for out-of-band data (TCP, for example).
STREAM_SHUT_RD	Used with stream_socket_shutdown() to disable further receptions. Added in PHP 5.2.1.

STREAM_SHUT_WR	Used with stream_socket_shutdown() to disable further transmissions. Added in PHP 5.2.1.
STREAM_SHUT_RDWR	Used with stream_socket_shutdown() to disable further receptions and transmissions. Added in PHP 5.2.1.

Note
The constants marked with * are just available since PHP 5.0.0.

Note
The constants marked with + are available since PHP 5.1.0 and are meant to be used with stream_socket_pair(). Please note that some of these constants might not be available in your system.

Stream Filters

A *filter* is a final piece of code which may perform operations on data as it is being read from or written to a stream. Any number of filters may be stacked onto a stream. Custom filters can be defined in a PHP script using stream_filter_register() or in an extension using the API Reference in Working with streams. To access the list of currently registered filters, use stream_get_filters().

Stream Contexts

A *context* is a set of *parameters* and wrapper specific *options* which modify or enhance the behavior of a stream. *Contexts* are created using stream_context_create() and can be passed to most filesystem related stream creation functions (i.e. fopen(), file(), file_get_contents(), etc...).

Options can be specified when calling stream_context_create(), or later using stream_context_set_option(). A list of wrapper specific *options* can be found in the Context options and parameters chapter.

Parameters can be specified for *contexts* using the stream_context_set_params() function.

Stream Errors

As with any file or socket related function, an operation on a stream may fail for a variety of normal reasons (i.e.: Unable to connect to remote host, file not found, etc...). A stream related call may also fail because the desired stream is not registered on the running system. See the array returned by stream_get_wrappers() for a list of streams supported by your installation of PHP. As with most PHP internal functions if a failure occurs an **E_WARNING** message will be generated describing the nature of the error.

Examples

Example #1 - Using file_get_contents() to retrieve data from multiple sources

```
<?php
/* Read local file from /home/bar */
$localfile = file_get_contents("/home/bar/foo.txt");

/* Identical to above, explicitly naming FILE scheme */
$localfile = file_get_contents("file:///home/bar/foo.txt");

/* Read remote file from www.example.com using HTTP */
$httpfile  = file_get_contents("http://www.example.com/foo.txt");

/* Read remote file from www.example.com using HTTPS */
$httpsfile = file_get_contents("https://www.example.com/foo.txt");

/* Read remote file from ftp.example.com using FTP */
$ftpfile   = file_get_contents("ftp://user:pass@ftp.example.com/foo.txt");

/* Read remote file from ftp.example.com using FTPS */
$ftpsfile  = file_get_contents("ftps://user:pass@ftp.example.com/foo.txt");
?>
```

Example #2 - Making a POST request to an https server

```
<?php
/* Send POST request to https://secure.example.com/form_action.php
* Include form elements named "foo" and "bar" with dummy values
*/

$sock = fsockopen("ssl://secure.example.com", 443, $errno, $errstr, 30);
if (!$sock) die("$errstr ($errno)\n");

$data = "foo=" . urlencode("Value for Foo") . "&bar=" . urlencode("Value for
Bar");

fwrite($sock, "POST /form_action.php HTTP/1.0\r\n");
fwrite($sock, "Host: secure.example.com\r\n");
fwrite($sock, "Content-type: application/x-www-form-urlencoded\r\n");
fwrite($sock, "Content-length: " . strlen($data) . "\r\n");
fwrite($sock, "Accept: */*\r\n");
fwrite($sock, "\r\n");
fwrite($sock, "$data\r\n");
fwrite($sock, "\r\n");

$headers = "";
while ($str = trim(fgets($sock, 4096)))
$headers .= "$str\n";
```

```
echo "\n";

$body = "";
while (!feof($sock))
$body .= fgets($sock, 4096);

fclose($sock);
?>
```

Example #3 - Writing data to a compressed file

```
<?php
/* Create a compressed file containing an arbitrarty string
* File can be read back using compress.zlib stream or just
* decompressed from the command line using 'gzip -d foo-bar.txt.gz'
*/
$fp = fopen("compress.zlib://foo-bar.txt.gz", "wb");
if (!$fp) die("Unable to create file.");

fwrite($fp, "This is a test.\n");

fclose($fp);
?>
```

Stream Functions

stream_bucket_append

stream_bucket_append -- Append bucket to brigade

Description

void **stream_bucket_append** (resource `$brigade`, resource `$bucket`)

Warning
This function is currently not documented; only its argument list is available.

stream_bucket_make_writeable

stream_bucket_make_writeable -- Return a bucket object from the brigade for operating on

Description

object **stream_bucket_make_writeable** (resource `$brigade`)

Warning
This function is currently not documented; only its argument list is available.

stream_bucket_new

stream_bucket_new -- Create a new bucket for use on the current stream

Description

object **stream_bucket_new** (resource `$stream`, string `$buffer`)

Warning
This function is currently not documented; only its argument list is available.

stream_bucket_prepend

stream_bucket_prepend -- Prepend bucket to brigade

Description

void **stream_bucket_prepend** (resource `$brigade`, resource `$bucket`)

Warning
This function is currently not documented; only its argument list is available.

stream_context_create

stream_context_create -- Create a streams context

Description

resource **stream_context_create** ([array `$options` [, array `$params`]])

Creates and returns a stream context with any options supplied in *`options`* preset.

Parameters

`options`
: Must be an associative array of associative arrays in the format *$arr['wrapper']['option'] = $value*. Default to an empty array.

`params`
: Must be an associative array in the format *$arr['parameter'] = $value*. Refer to stream_context_set_params() for a listing of standard stream parameters.

Return Values

A stream context resource.

ChangeLog

Version	Description
5.3.0	Added the optional *`params`* argument.

Examples

Example #4 - Using stream_context_create()

```
<?php
$opts = array(
 'http'=>array(
   'method'=>"GET",
   'header'=>"Accept-language: en\r\n" .
             "Cookie: foo=bar\r\n"
 )
);
```

```
$context = stream_context_create($opts);

/* Sends an http request to www.example.com
  with additional headers shown above */
$fp = fopen('http://www.example.com', 'r', false, $context);
fpassthru($fp);
fclose($fp);
?>
```

See Also

- stream_context_set_option()
- Listing of supported wrappers (List of Supported Protocols/Wrappers)
- Context options (Context options and parameters)

stream_context_get_default

stream_context_get_default -- Retreive the default streams context

Description

resource **stream_context_get_default** ([array `$options`])

Returns the default stream context which is used whenever file operations (fopen(), file_get_contents(), etc...) are called without a context parameter. Options for the default context can optionally be specified with this function using the same syntax as stream_context_create().

`options` must be an associative array of associative arrays in the format *$arr['wrapper']['option'] = $value*.

Example #5 - Using stream_context_get_default()

```
<?php
$default_opts = array(
 'http'=>array(
   'method'=>"GET",
   'header'=>"Accept-language: en\r\n" .
             "Cookie: foo=bar",
   'proxy'=>"tcp://10.54.1.39:8000"
 )
);


$alternate_opts = array(
 'http'=>array(
   'method'=>"POST",
   'header'=>"Content-type: application/x-www-form-urlencoded\r\n" .
             "Content-length: " . strlen("baz=bomb"),
   'content'=>"baz=bomb"
 )
);

$default = stream_context_get_default($default_opts);
$alternate = stream_context_create($alternate_opts);

/* Sends a regular GET request to proxy server at 10.54.1.39
* For www.example.com using context options specified in $default_opts
*/
readfile('http://www.example.com');

/* Sends a POST request directly to www.example.com
* Using context options specified in $alternate_opts
*/
readfile('http://www.example.com', false, $alternate);

?>
```

See also stream_context_create(), and Listing of supported wrappers with context options (List of Supported Protocols/Wrappers).

stream_context_get_options

stream_context_get_options -- Retrieve options for a stream/wrapper/context

Description

array **stream_context_get_options** (resource `$stream_or_context`)

Returns an array of options on the specified stream or context.

stream_context_set_option

stream_context_set_option -- Sets an option for a stream/wrapper/context

Description

bool **stream_context_set_option** (resource `$stream_or_context`, string `$wrapper`, string `$option`, mixed `$value`)

bool **stream_context_set_option** (resource `$stream_or_context`, array `$options`)

Sets an option on the specified context. *`value`* is set to *`option`* for *`wrapper`*

stream_context_set_params

stream_context_set_params -- Set parameters for a stream/wrapper/context

Description

bool **stream_context_set_params** (resource `$stream_or_context`, array `$params`)

`params` should be an associative array of the structure: *$params['paramname'] = "paramvalue";*.

Parameters

Parameters	**Purpose**
notification	Name of user-defined callback function to be called whenever a stream triggers a notification.
options	Array of options as in stream_context_create().

stream_copy_to_stream

stream_copy_to_stream -- Copies data from one stream to another

Description

int **stream_copy_to_stream** (resource `$source`, resource `$dest` [, int `$maxlength` [, int `$offset`]])

Makes a copy of up to *`maxlength`* bytes of data from the current position (or from the *`offset`* position, if specified) in *`source`* to *`dest`*. If *`maxlength`* is not specified, all remaining content in *`source`* will be copied.

Parameters

`source`
: The source stream

`dest`
: The destination stream

`maxlength`
: Maximum bytes to copy

`offset`
: The offset where to start to copy data

Return Values

Returns the total count of bytes copied.

ChangeLog

Version	Description
5.1.0	Added the *`offset`* parameter

Examples

Example #6 - A stream_copy_to_stream() example

```
<?php
$src = fopen('http://www.example.com', 'r');
$dest1 = fopen('first1k.txt', 'w');
$dest2 = fopen('remainder.txt', 'w');

echo stream_copy_to_stream($src, $dest1, 1024) . " bytes copied to
first1k.txt\n";
echo stream_copy_to_stream($src, $dest2) . " bytes copied to
remainder.txt\n";

?>
```

See Also

- copy()

stream_encoding

stream_encoding -- Set character set for stream encoding

Description

bool **stream_encoding** (resource `$stream` [, string `$encoding`])

Warning
This function is currently not documented; only its argument list is available.

stream_filter_append

stream_filter_append -- Attach a filter to a stream

Description

resource **stream_filter_append** (resource `$stream`, string `$filtername` [, int `$read_write` [, mixed `$params`]])

Adds *`filtername`* to the list of filters attached to *`stream`*. This filter will be added with the specified *`params`* to the *end* of the list and will therefore be called last during stream operations. To add a filter to the beginning of the list, use stream_filter_prepend().

By default, stream_filter_append() will attach the filter to the *read filter chain* if the file was opened for reading (i.e. File Mode: *r*, and/or +). The filter will also be attached to the *write filter chain* if the file was opened for writing (i.e. File Mode: *w*, *a*, and/or +). **STREAM_FILTER_READ**, **STREAM_FILTER_WRITE**, and/or **STREAM_FILTER_ALL** can also be passed to the *`read_write`* parameter to override this behavior.

As of PHP 5.1.0, this function returns a resource which can be used to refer to this filter instance during a call to stream_filter_remove(). Prior to PHP 5.1.0, this function returns **TRUE** on success or **FALSE** on failure.

Example #7 - Controlling where filters are applied

```
<?php
/* Open a test file for reading and writing */
$fp = fopen('test.txt', 'w+');

/* Apply the ROT13 filter to the
* write filter chain, but not the
* read filter chain */
stream_filter_append($fp, "string.rot13", STREAM_FILTER_WRITE);

/* Write a simple string to the file
* it will be ROT13 transformed on the
* way out */
fwrite($fp, "This is a test\n");

/* Back up to the beginning of the file */
rewind($fp);

/* Read the contents of the file back out.
* Had the filter been applied to the
* read filter chain as well, we would see
* the text ROT13ed back to its original state */
fpassthru($fp);

fclose($fp);

/* Expected Output
```

```
  ---------------

Guvf vf n grfg

*/
?>
```

Note

When using custom (user) filters

stream_filter_register() must be called first in order to register the desired user filter to *filtername*.

Note

Stream data is read from resources (both local and remote) in chunks, with any unconsumed data kept in internal buffers. When a new filter is appended to a stream, data in the internal buffers is processed through the new filter at that time. This differs from the behavior of stream_filter_prepend().

See also stream_filter_register(), stream_filter_prepend(), and stream_get_filters().

stream_filter_prepend

stream_filter_prepend -- Attach a filter to a stream

Description

resource **stream_filter_prepend** (resource `$stream`, string `$filtername` [, int `$read_write` [, mixed `$params`]])

Adds *`filtername`* to the list of filters attached to *`stream`*. This filter will be added with the specified *`params`* to the *beginning* of the list and will therefore be called first during stream operations. To add a filter to the end of the list, use stream_filter_append().

By default, stream_filter_prepend() will attach the filter to the *read filter chain* if the file was opened for reading (i.e. File Mode: *r*, and/or +). The filter will also be attached to the *write filter chain* if the file was opened for writing (i.e. File Mode: *w*, *a*, and/or +). **STREAM_FILTER_READ**, **STREAM_FILTER_WRITE**, and/or **STREAM_FILTER_ALL** can also be passed to the *`read_write`* parameter to override this behavior. See stream_filter_append() for an example of using this parameter.

As of PHP 5.1.0, this function returns a resource which can be used to refer to this filter instance during a call to stream_filter_remove(). Prior to PHP 5.1.0, this function returns **TRUE** on success or **FALSE** on failure.

Note
When using custom (user) filters stream_filter_register() must be called first in order to register the desired user filter to *`filtername`*.

Note
Stream data is read from resources (both local and remote) in chunks, with any unconsumed data kept in internal buffers. When a new filter is prepended to a stream, data in the internal buffers, which has already been processed through other filters will *not* be reprocessed through the new filter at that time. This differs from the behavior of stream_filter_append().

See also stream_filter_register(), and stream_filter_append().

stream_filter_register

stream_filter_register -- Register a stream filter implemented as a PHP class derived from *php_user_filter*

Description

bool **stream_filter_register** (string `$filtername`, string `$classname`)

stream_filter_register() allows you to implement your own filter on any registered stream used with all the other filesystem functions (such as fopen(), fread() etc.).

To implement a filter, you need to define a class as an extension of *php_user_filter* with a number of member functions as defined below. When performing read/write operations on the stream to which your filter is attached, PHP will pass the data through your filter (and any other filters attached to that stream) so that the data may be modified as desired. You must implement the methods exactly as described below - doing otherwise will lead to undefined behaviour.

stream_filter_register() will return **FALSE** if the *`filtername`* is already defined.

int **filter** (resource `$in`, resource `$out`, int `&$consumed`, bool `$closing`)

This method is called whenever data is read from or written to the attached stream (such as with fread() or fwrite()). *`in`* is a resource pointing to a *bucket brigade* which contains one or more *bucket* objects containing data to be filtered. *`out`* is a resource pointing to a second *bucket brigade* into which your modified buckets should be placed. *`consumed`*, which must *always* be declared by reference, should be incremented by the length of the data which your filter reads in and alters. In most cases this means you will increment *`consumed`* by $bucket->datalen for each $bucket. If the stream is in the process of closing (and therefore this is the last pass through the filterchain), the *`closing`* parameter will be set to **TRUE** The **filter** method must return one of three values upon completion.

Return Value	**Meaning**
PSFS_PASS_ON	Filter processed successfully with data available in the *`out`* *bucket brigade*.
PSFS_FEED_ME	Filter processed successfully, however no data was available to return. More data is required from the stream or prior filter.
PSFS_ERR_FATAL (default)	The filter experienced an unrecoverable error and cannot continue.

bool **onCreate** (void)

This method is called during instantiation of the filter class object. If your filter allocates or initializes any other resources (such as a buffer), this is the place to do it. Your implementation of this method should return **FALSE** on failure, or **TRUE** on success.

When your filter is first instantiated, and *yourfilter->onCreate()* is called, a number of properties will be available as shown in the table below.

Property	**Contents**
FilterClass->filtername	A string containing the name the filter was instantiated with. Filters may be registered under multiple names or under wildcards. Use this property to determine which name was used.
FilterClass->params	The contents of the `params` parameter passed to stream_filter_append() or stream_filter_prepend().

void **onClose** (void)

This method is called upon filter shutdown (typically, this is also during stream shutdown), and is executed *after* the *flush* method is called. If any resources were allocated or initialzed during *onCreate* this would be the time to destroy or dispose of them.

The example below implements a filter named *strtoupper* on the *foo-bar.txt* stream which will capitalize all letter characters written to/read from that stream.

Example #8 - Filter for capitalizing characters on foo-bar.txt stream

```
<?php

/* Define our filter class */
class strtoupper_filter extends php_user_filter {
 function filter($in, $out, &$consumed, $closing)
 {
   while ($bucket = stream_bucket_make_writeable($in)) {
     $bucket->data = strtoupper($bucket->data);
     $consumed += $bucket->datalen;
     stream_bucket_append($out, $bucket);
   }
   return PSFS_PASS_ON;
 }
}

/* Register our filter with PHP */
stream_filter_register("strtoupper", "strtoupper_filter")
   or die("Failed to register filter");

$fp = fopen("foo-bar.txt", "w");
```

```
/* Attach the registered filter to the stream just opened */
stream_filter_append($fp, "strtoupper");

fwrite($fp, "Line1\n");
fwrite($fp, "Word - 2\n");
fwrite($fp, "Easy As 123\n");

fclose($fp);

/* Read the contents back out
*/
readfile("foo-bar.txt");

?>
```

The above example will output:

```
LINE1
WORD - 2
EASY AS 123
```

Example #9 - Registering a generic filter class to match multiple filter names.

```
<?php

/* Define our filter class */
class string_filter extends php_user_filter {
 var $mode;

 function filter($in, $out, &$consumed, $closing)
 {
   while ($bucket = stream_bucket_make_writeable($in)) {
     if ($this->mode == 1) {
       $bucket->data = strtoupper($bucket->data);
     } elseif ($this->mode == 0) {
       $bucket->data = strtolower($bucket->data);
     }

     $consumed += $bucket->datalen;
     stream_bucket_append($out, $bucket);
   }
   return PSFS_PASS_ON;
 }

 function onCreate()
 {
   if ($this->filtername == 'str.toupper') {
     $this->mode = 1;
   } elseif ($this->filtername == 'str.tolower') {
     $this->mode = 0;
   } else {
     /* Some other str.* filter was asked for,
        report failure so that PHP will keep looking */
     return false;
   }
```

```
   return true;
 }
}

/* Register our filter with PHP */
stream_filter_register("str.*", "string_filter")
   or die("Failed to register filter");

$fp = fopen("foo-bar.txt", "w");

/* Attach the registered filter to the stream just opened
  We could alternately bind to str.tolower here */
stream_filter_append($fp, "str.toupper");

fwrite($fp, "Line1\n");
fwrite($fp, "Word - 2\n");
fwrite($fp, "Easy As 123\n");

fclose($fp);

/* Read the contents back out
*/
readfile("foo-bar.txt");
?>
```

The above example will output:

```
LINE1
WORD - 2
EASY AS 123
```

See also stream_wrapper_register(), stream_filter_prepend(), and stream_filter_append().

stream_filter_remove

stream_filter_remove -- Remove a filter from a stream

Description

bool **stream_filter_remove** (resource `$stream_filter`)

Removes a stream filter previously added to a stream with stream_filter_prepend() or stream_filter_append(). Any data remaining in the filter's internal buffer will be flushed through to the next filter before removing it.

Example #10 - Dynamicly refiltering a stream

```
<?php
/* Open a test file for reading and writing */
$fp = fopen("test.txt", "rw");

$rot13_filter = stream_filter_append($fp, "string.rot13",
STREAM_FILTER_WRITE);
fwrite($fp, "This is ");
stream_filter_remove($rot13_filter);
fwrite($fp, "a test\n");

rewind($fp);
fpassthru($fp);
fclose($fp);

/* Expected Output
  ---------------

Guvf vf a test

*/
?>
```

See also stream_filter_register(), stream_filter_append(), and stream_filter_prepend().

stream_get_contents

stream_get_contents -- Reads remainder of a stream into a string

Description

string **stream_get_contents** (resource `$handle` [, int `$maxlength` [, int `$offset`]])

Identical to file_get_contents(), except that stream_get_contents() operates on an already open stream resource and returns the remaining contents in a string, up to *`maxlength`* bytes and starting at the specified *`offset`*.

Parameters

`handle` (resource)
A stream resource (e.g. returned from fopen())

`maxlength` (integer)
The maximum bytes to read. Defaults to -1 (read all the remaining buffer).

`offset` (integer)
Seek to the specified offset before reading. Added in PHP 5.1.0.

Return Values

Returns a string, or **FALSE** on failure.

Examples

Example #11 - stream_get_contents() example

```
<?php

if ($stream = fopen('http://www.example.com', 'r')) {
   // print all the page starting at the offset 10
   echo stream_get_contents($stream, -1, 10);

   fclose($stream);
}


if ($stream = fopen('http://www.example.net', 'r')) {
   // print the first 5 bytes
   echo stream_get_contents($stream, 5);

   fclose($stream);
}
```

```
?>
```

See Also

- fgets()
- fread()
- fpassthru()

Note
This function is binary-safe.

stream_get_filters

stream_get_filters -- Retrieve list of registered filters

Description

array **stream_get_filters** (void)

Returns an indexed array containing the name of all stream filters available on the running system.

Example #12 - Using stream_get_filters()

```
<?php
$streamlist = stream_get_filters();
print_r($streamlist);
?>
```

Output will be similar to the following. Note: there may be more or fewer filters in your version of PHP.

```
Array (
 [0] => string.rot13
 [1] => string.toupper
 [2] => string.tolower
 [3] => string.base64
 [4] => string.quoted-printable
)
```

See also stream_filter_register(), and stream_get_wrappers().

stream_get_line

stream_get_line -- Gets line from stream resource up to a given delimiter

Description

string **stream_get_line** (resource `$handle`, int `$length` [, string `$ending`])

Returns a string of up to *`length`* bytes read from the file pointed to by *`handle`*. Reading ends when *`length`* bytes have been read, when the string specified by *`ending`* is found (which is *not* included in the return value), or on EOF (whichever comes first).

If an error occurs, returns **FALSE**.

This function is nearly identical to fgets() except in that it allows end of line delimiters other than the standard \n, \r, and \r\n, and does *not* return the delimiter itself.

See also fread(), fgets(), and fgetc().

stream_get_meta_data

stream_get_meta_data -- Retrieves header/meta data from streams/file pointers

Description

array **stream_get_meta_data** (resource `$stream`)

Returns information about an existing *`stream`*. The stream can be any stream created by fopen(), fsockopen() and pfsockopen(). The result array contains the following items:

- *`timed_out`* (bool) - **TRUE** if the stream timed out while waiting for data on the last call to fread() or fgets().
- *`blocked`* (bool) - **TRUE** if the stream is in blocking IO mode. See stream_set_blocking() .
- *`eof`* (bool) - **TRUE** if the stream has reached end-of-file. Note that for socket streams this member can be **TRUE** even when *`unread_bytes`* is non-zero. To determine if there is more data to be read, use feof() instead of reading this item.
- *`unread_bytes`* (int) - the number of bytes currently contained in the PHP's own internal buffer.

Note
You shouldn't use this value in a script.

The following items were added in PHP 4.3.0:

- *`stream_type`* (string) - a label describing the underlying implementation of the stream.
- *`wrapper_type`* (string) - a label describing the protocol wrapper implementation layered over the stream. See List of Supported Protocols/Wrappers for more information about wrappers.
- *`wrapper_data`* (mixed) - wrapper specific data attached to this stream. See List of Supported Protocols/Wrappers for more information about wrappers and their wrapper data.
- *`filters`* (array) - and array containing the names of any filters that have been stacked onto this stream. Documentation on filters can be found in the Filters appendix.

Note
This function was introduced in PHP 4.3.0, but prior to this version,

socket_get_status() could be used to retrieve the first four items, for *socket based streams only*.

In PHP 4.3.0 and later, socket_get_status() is an alias for this function.

Note

This function does NOT work on sockets created by the Socket extension.

The following items were added in PHP 5.0.0:

- *mode* (string) - the type of access required for this stream (see Table 1 of the fopen() reference)
- *seekable* (bool) - whether the current stream can be seeked.
- *uri* (string) - the URI/filename associated with this stream.

stream_get_transports

stream_get_transports -- Retrieve list of registered socket transports

Description

array **stream_get_transports** (void)

Returns an indexed array containing the name of all socket transports available on the running system.

Example #13 - Using stream_get_transports()

```
<?php
$xportlist = stream_get_transports();
print_r($xportlist);
?>
```

Output will be similar to the following. Note: there may be more or fewer transports in your version of PHP.

```
Array (
 [0] => tcp
 [1] => udp
 [2] => unix
 [3] => udg
)
```

See also stream_get_filters(), and stream_get_wrappers().

stream_get_wrappers

stream_get_wrappers -- Retrieve list of registered streams

Description

array **stream_get_wrappers** (void)

Returns an indexed array containing the name of all stream wrappers available on the running system.

Example #14 - stream_get_wrappers() example

```
<?php
print_r(stream_get_wrappers());
?>
```

The above example will output something similar to:

```
Array
(
   [0] => php
   [1] => file
   [2] => http
   [3] => ftp
   [4] => compress.bzip2
   [5] => compress.zlib
)
```

Example #15 - Checking for the existence of a stream wrapper

```
<?php
// check for the existence of the bzip2 stream wrapper
if (in_array('compress.bzip2', stream_get_wrappers())) {
   echo 'compress.bzip2:// support enabled.';
} else {
   echo 'compress.bzip2:// support not enabled.';
}
?>
```

See also stream_wrapper_register().

stream_notification_callback

stream_notification_callback -- A callback function for the *notification* context paramater

Description

void **stream_notification_callback** (int `$notification_code`, int `$severity`, string `$message`, int `$message_code`, int `$bytes_transferred`, int `$bytes_max`)

A callback function called during an event.

Note
This is *not* a real function, only a prototype of how the function should be.

Parameters

`notification_code`
: One of the **STREAM_NOTIFY_*** notification constants.

`severity`
: One of the **STREAM_NOTIFY_SEVERITY_*** notification constants.

`message`
: Passed if a descriptive message is available for the event.

`message_code`
: Passed if a descriptive message code is available for the event. The meaning of this value is dependent on the specific wrapper in use.

`bytes_transferred`
: If applicable, the *`bytes_transferred`* will be populated.

`bytes_max`
: If applicable, the *`bytes_max`* will be populated.

Return Values

No value is returned.

Examples

Example #16 - stream_notification_callback() example

```
<?php
function stream_notification_callback($notification_code, $severity,
$message, $message_code, $bytes_transferred, $bytes_max) {

   switch($notification_code) {
       case STREAM_NOTIFY_RESOLVE:
       case STREAM_NOTIFY_AUTH_REQUIRED:
       case STREAM_NOTIFY_COMPLETED:
       case STREAM_NOTIFY_FAILURE:
       case STREAM_NOTIFY_AUTH_RESULT:
           var_dump($notification_code, $severity, $message, $message_code,
$bytes_transferred, $bytes_max);
           /* Ignore */
           break;

       case STREAM_NOTIFY_REDIRECTED:
           echo "Being redirected to: ", $message;
           break;

       case STREAM_NOTIFY_CONNECT:
           echo "Conntected...";
           break;

       case STREAM_NOTIFY_FILE_SIZE_IS:
           echo "Got the filesize: ", $bytes_max;
           break;

       case STREAM_NOTIFY_MIME_TYPE_IS:
           echo "Found the mime-type: ", $message;
           break;

       case STREAM_NOTIFY_PROGRESS:
           echo "Made some progress, downloaded ", $bytes_transferred, " so
far";
           break;
   }
   echo "\n";
}

$ctx = stream_context_create(null, array("notification" =>
"stream_notification_callback"));

file_get_contents("http://php.net/contact", false, $ctx);
?>
```

The above example will output something similar to:

```
Conntected...
Found the mime-type: text/html; charset=utf-8
Being redirected to: http://no.php.net/contact
Conntected...
Got the filesize: 0
Found the mime-type: text/html; charset=utf-8
Being redirected to: http://no.php.net/contact.php
Conntected...
Got the filesize: 4589
Found the mime-type: text/html;charset=utf-8
```

```
Made some progress, downloaded 0 so far
Made some progress, downloaded 0 so far
Made some progress, downloaded 0 so far
Made some progress, downloaded 1440 so far
Made some progress, downloaded 2880 so far
Made some progress, downloaded 4320 so far
Made some progress, downloaded 5760 so far
Made some progress, downloaded 6381 so far
Made some progress, downloaded 7002 so far
```

Example #17 - Simple progressbar for commandline download client

```
<?php
function usage($argv) {
   echo "Usage:\n";
   printf("\tphp %s <http://example.com/file> <localfile>\n", $argv[0]);
   exit(1);
}

function stream_notification_callback($notification_code, $severity,
$message, $message_code, $bytes_transferred, $bytes_max) {
   static $filesize = null;

   switch($notification_code) {
   case STREAM_NOTIFY_RESOLVE:
   case STREAM_NOTIFY_AUTH_REQUIRED:
   case STREAM_NOTIFY_COMPLETED:
   case STREAM_NOTIFY_FAILURE:
   case STREAM_NOTIFY_AUTH_RESULT:
       /* Ignore */
       break;

   case STREAM_NOTIFY_REDIRECTED:
       echo "Being redirected to: ", $message, "\n";
       break;

   case STREAM_NOTIFY_CONNECT:
       echo "Conntected...\n";
       break;

   case STREAM_NOTIFY_FILE_SIZE_IS:
       $filesize = $bytes_max;
       echo "Filesize: ", $filesize, "\n";
       break;

   case STREAM_NOTIFY_MIME_TYPE_IS:
       echo "Mime-type: ", $message, "\n";
       break;

   case STREAM_NOTIFY_PROGRESS:
       if ($bytes_transferred > 0) {
           if (!isset($filesize)) {
               printf("\rUnknown filesize.. %2d kb done..",
$bytes_transferred/1024);
           } else {
```

```
                $length = (int)(($bytes_transferred/$filesize)*100);
                printf("\r[%-100s] %d%% (%2d/%2d kb)", str_repeat("=",
$length). ">", $length, ($bytes_transferred/1024), $filesize/1024);
             }
        }
        break;
    }
}

isset($argv[1], $argv[2]) or usage($argv);

$ctx = stream_context_create(null, array("notification" =>
"stream_notification_callback"));

$fp = fopen($argv[1], "r", false, $ctx);
if (is_resource($fp) && file_put_contents($argv[2], $fp)) {
   echo "\nDone!\n";
   exit(0);
}

$err = error_get_last();
echo "\nErrrrrorr..\n", $err["message"], "\n";
exit(1);
?>
```

Executing the example above with: *php -n fetch.php http://no2.php.net/get/php-5-LATEST.tar.bz2/from/this/mirror php-latest.tar.bz2* will output something similar too:

```
Conntected...
Mime-type: text/html; charset=utf-8
Being redirected to: http://no2.php.net/distributions/php-5.2.5.tar.bz2
Conntected...
Filesize: 7773024
Mime-type: application/octet-stream
[=======================================>
                      ] 40% (3076/7590 kb)
```

See Also

- Context parameters

stream_register_wrapper

stream_register_wrapper -- Alias of stream_wrapper_register()

Description

This function is an alias of: stream_wrapper_register().

stream_resolve_include_path

stream_resolve_include_path -- Determine what file will be opened by calls to fopen() with a relative path

Description

string **stream_resolve_include_path** (string `$filename` [, resource `$context`])

Warning
This function is currently not documented; only its argument list is available.

stream_select

stream_select -- Runs the equivalent of the select() system call on the given arrays of streams with a timeout specified by tv_sec and tv_usec

Description

int **stream_select** (array `&$read`, array `&$write`, array `&$except`, int `$tv_sec` [, int `$tv_usec`])

The stream_select() function accepts arrays of streams and waits for them to change status. Its operation is equivalent to that of the socket_select() function except in that it acts on streams.

The streams listed in the *`read`* array will be watched to see if characters become available for reading (more precisely, to see if a read will not block - in particular, a stream resource is also ready on end-of-file, in which case an fread() will return a zero length string).

The streams listed in the *`write`* array will be watched to see if a write will not block.

The streams listed in the *`except`* array will be watched for high priority exceptional ("out-of-band") data arriving.

Note
When stream_select() returns, the arrays *`read`*, *`write`* and *`except`* are modified to indicate which stream resource(s) actually changed status.

The *`tv_sec`* and *`tv_usec`* together form the *timeout* parameter, *`tv_sec`* specifies the number of seconds while *`tv_usec`* the number of microseconds. The *timeout* is an upper bound on the amount of time that stream_select() will wait before it returns. If *`tv_sec`* and *`tv_usec`* are both set to *0*, stream_select() will not wait for data - instead it will return immediately, indicating the current status of the streams. If *`tv_sec`* is **NULL** stream_select() can block indefinitely, returning only when an event on one of the watched streams occurs (or if a signal interrupts the system call).

On success stream_select() returns the number of stream resources contained in the modified arrays, which may be zero if the timeout expires before anything interesting happens. On error **FALSE** is returned and a warning raised (this can happen if the system call is interrupted by an incoming signal).

Warning
Using a timeout value of *0* allows you to instantaneously poll the status of the streams, however, it is NOT a good idea to use a *0* timeout value in a loop as it will cause your script to consume too much CPU time.

It is much better to specify a timeout value of a few seconds, although if you need to be checking and running other code concurrently, using a timeout value of at least *200000* microseconds will help reduce the CPU usage of your script.

Remember that the timeout value is the maximum time that will elapse; stream_select() will return as soon as the requested streams are ready for use.

You do not need to pass every array to stream_select(). You can leave it out and use an empty array or **NULL** instead. Also do not forget that those arrays are passed *by reference* and will be modified after stream_select() returns.

This example checks to see if data has arrived for reading on either `$stream1` or `$stream2` . Since the timeout value is *0* it will return immediately:

```
<?php
/* Prepare the read array */
$read   = array($stream1, $stream2);
$write  = NULL;
$except = NULL;
if (false === ($num_changed_streams = stream_select($read, $write, $except, 0)))
{
   /* Error handling */
} elseif ($num_changed_streams > 0) {
   /* At least on one of the streams something interesting happened */
}
?>
```

Note

Due to a limitation in the current Zend Engine it is not possible to pass a constant modifier like **NULL** directly as a parameter to a function which expects this parameter to be passed by reference. Instead use a temporary variable or an expression with the leftmost member being a temporary variable:

```
<?php
$e = NULL;
stream_select($r, $w, $e, 0);
?>
```

Note

Be sure to use the === operator when checking for an error. Since the stream_select() may return 0 the comparison with == would evaluate to **TRUE**:

```
<?php
$e = NULL;
if (false === stream_select($r, $w, $e, 0)) {
   echo "stream_select() failed\n";
}
?>
```

Note

If you read/write to a stream returned in the arrays be aware that they do not necessarily read/write the full amount of data you have requested. Be prepared to even only be able to read/write a single byte.

Note

Windows compatibility: stream_select() used on a pipe returned from proc_open() may cause data loss under Windows 98.

Use of stream_select() on file descriptors returned by proc_open() will fail and return **FALSE** under Windows.

See also stream_set_blocking().

stream_set_blocking

stream_set_blocking -- Set blocking/non-blocking mode on a stream

Description

bool **stream_set_blocking** (resource `$stream`, int `$mode`)

If *`mode`* is 0, the given stream will be switched to non-blocking mode, and if 1, it will be switched to blocking mode. This affects calls like fgets() and fread() that read from the stream. In non-blocking mode an fgets() call will always return right away while in blocking mode it will wait for data to become available on the stream.

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

This function was previously called as **set_socket_blocking()** and later socket_set_blocking() but this usage is deprecated.

Note
Prior to PHP 4.3, this function only worked on socket based streams. Since PHP 4.3, this function works for any stream that supports non-blocking mode (currently, regular files and socket streams).

See also stream_select().

stream_set_timeout

stream_set_timeout -- Set timeout period on a stream

Description

bool **stream_set_timeout** (resource `$stream`, int `$seconds` [, int `$microseconds`])

Sets the timeout value on *`stream`*, expressed in the sum of *`seconds`* and *`microseconds`*. Returns **TRUE** on success or **FALSE** on failure.

When the stream times out, the 'timed_out' key of the array returned by stream_get_meta_data() is set to **TRUE**, although no error/warning is generated.

Example #18 - stream_set_timeout() **example**

```
<?php
$fp = fsockopen("www.example.com", 80);
if (!$fp) {
   echo "Unable to open\n";
} else {

   fwrite($fp, "GET / HTTP/1.0\r\n\r\n");
   stream_set_timeout($fp, 2);
   $res = fread($fp, 2000);

   $info = stream_get_meta_data($fp);
   fclose($fp);

   if ($info['timed_out']) {
       echo 'Connection timed out!';
   } else {
       echo $res;
   }

}
?>
```

Note

As of PHP 4.3, this function can (potentially) work on any kind of stream. In PHP 4.3, socket based streams are still the only kind supported in the PHP core, although streams from other extensions may support this function.

Note
This function doesn't work with advanced operations like stream_socket_recvfrom(), use stream_select() with timeout parameter instead.

This function was previously called as **set_socket_timeout()** and later socket_set_timeout() but this usage is deprecated.

See also fsockopen() and fopen().

stream_set_write_buffer

stream_set_write_buffer -- Sets file buffering on the given stream

Description

int **stream_set_write_buffer** (resource `$stream`, int `$buffer`)

Output using fwrite() is normally buffered at 8K. This means that if there are two processes wanting to write to the same output stream (a file), each is paused after 8K of data to allow the other to write. stream_set_write_buffer() sets the buffering for write operations on the given filepointer *`stream`* to *`buffer`* bytes. If *`buffer`* is 0 then write operations are unbuffered. This ensures that all writes with fwrite() are completed before other processes are allowed to write to that output stream.

The function returns 0 on success, or EOF if the request cannot be honored.

The following example demonstrates how to use stream_set_write_buffer() to create an unbuffered stream.

Example #19 - stream_set_write_buffer() **example**

```
<?php
$fp = fopen($file, "w");
if ($fp) {
 stream_set_write_buffer($fp, 0);
 fwrite($fp, $output);
 fclose($fp);
}
?>
```

See also fopen() and fwrite().

stream_socket_accept

stream_socket_accept -- Accept a connection on a socket created by stream_socket_server()

Description

resource **stream_socket_accept** (resource `$server_socket` [, float `$timeout` [, string `&$peername`]])

Accept a connection on a socket previously created by stream_socket_server(). If *`timeout`* is specified, the default socket accept timeout will be overridden with the time specified in seconds. The name (address) of the client which connected will be passed back in *`peername`* if included and available from the selected transport.

`peername` can also be determined later using stream_socket_get_name().

If the call fails, it will return **FALSE**.

Warning
This function should not be used with UDP server sockets. Instead, use stream_socket_recvfrom() and stream_socket_sendto().

See also stream_socket_server(), stream_socket_get_name(), stream_set_blocking(), stream_set_timeout(), fgets(), fgetss(), fwrite(), fclose(), feof(), and the Curl extension.

stream_socket_client

stream_socket_client -- Open Internet or Unix domain socket connection

Description

resource **stream_socket_client** (string `$remote_socket` [, int `&$errno` [, string `&$errstr` [, float `$timeout` [, int `$flags` [, resource `$context`]]]]])

Initiates a stream or datagram connection to the destination specified by *`remote_socket`*. The type of socket created is determined by the transport specified using standard URL formatting: *transport://target*. For Internet Domain sockets (AF_INET) such as TCP and UDP, the *target* portion of the *`remote_socket`* parameter should consist of a hostname or IP address followed by a colon and a port number. For Unix domain sockets, the *`target`* portion should point to the socket file on the filesystem. The optional *`timeout`* can be used to set a timeout in seconds for the connect system call. *`flags`* is a bitmask field which may be set to any combination of connection flags. Currently the selection of connection flags is limited to **STREAM_CLIENT_CONNECT** (default), **STREAM_CLIENT_ASYNC_CONNECT** and **STREAM_CLIENT_PERSISTENT**.

Note
If you need to set a timeout for reading/writing data over the socket, use stream_set_timeout(), as the *`timeout`* parameter to stream_socket_client() only applies while connecting the socket.

Note
The timeout parameter only applies if you are not making an asynchronous connection attempt.

stream_socket_client() returns a stream resource which may be used together with the other file functions (such as fgets(), fgetss(), fwrite(), fclose(), and feof()).

If the call fails, it will return **FALSE** and if the optional *`errno`* and *`errstr`* arguments are present they will be set to indicate the actual system level error that occurred in the system-level *connect()* call. If the value returned in *`errno`* is *0* and the function returned **FALSE**, it is an indication that the error occurred before the *connect()* call. This is most likely due to a problem initializing the socket. Note that the *`errno`* and *`errstr`* arguments will always be passed by reference.

Depending on the environment, the Unix domain or the optional connect timeout may not be available. A list of available transports can be retrieved using stream_get_transports(). See List of Supported Socket Transports for a list of built in transports.

The stream will by default be opened in blocking mode. You can switch it to non-blocking mode by using stream_set_blocking().

Example #20 - stream_socket_client() Example

```
<?php
$fp = stream_socket_client("tcp://www.example.com:80", $errno, $errstr, 30);
if (!$fp) {
   echo "$errstr ($errno)<br />\n";
} else {
   fwrite($fp, "GET / HTTP/1.0\r\nHost: www.example.com\r\nAccept:
*/*\r\n\r\n");
   while (!feof($fp)) {
       echo fgets($fp, 1024);
   }
   fclose($fp);
}
?>
```

The example below shows how to retrieve the day and time from the UDP service "daytime" (port 13) in your own machine.

Example #21 - Using UDP connection

```
<?php
$fp = stream_socket_client("udp://127.0.0.1:13", $errno, $errstr);
if (!$fp) {
   echo "ERROR: $errno - $errstr<br />\n";
} else {
   fwrite($fp, "\n");
   echo fread($fp, 26);
   fclose($fp);
}
?>
```

Warning

UDP sockets will sometimes appear to have opened without an error, even if the remote host is unreachable. The error will only become apparent when you read or write data to/from the socket. The reason for this is because UDP is a "connectionless" protocol, which means that the operating system does not try to establish a link for the socket until it actually needs to send or receive data.

Note

When specifying a numerical IPv6 address (e.g. *fe80::1*), you must enclose the IP in square brackets?for example, *tcp:///[fe80::1]:80*.

See also stream_socket_server(), stream_set_blocking(), stream_set_timeout(),

stream_select(), fgets(), fgetss(), fwrite(), fclose(), feof(), and the Curl extension.

stream_socket_enable_crypto

stream_socket_enable_crypto -- Turns encryption on/off on an already connected socket

Description

mixed **stream_socket_enable_crypto** (resource `$stream`, bool `$enable` [, int `$crypto_type` [, resource `$session_stream`]])

When called with the *`crypto_type`* parameter, stream_socket_enable_crypto() will setup encryption on the stream using the specified method.

Valid values for *`crypto_type`*

- **STREAM_CRYPTO_METHOD_SSLv2_CLIENT**
- **STREAM_CRYPTO_METHOD_SSLv3_CLIENT**
- **STREAM_CRYPTO_METHOD_SSLv23_CLIENT**
- **STREAM_CRYPTO_METHOD_TLS_CLIENT**
- **STREAM_CRYPTO_METHOD_SSLv2_SERVER**
- **STREAM_CRYPTO_METHOD_SSLv3_SERVER**
- **STREAM_CRYPTO_METHOD_SSLv23_SERVER**
- **STREAM_CRYPTO_METHOD_TLS_SERVER**

Once the crypto settings are established, cryptography can be turned on and off dynamically by passing **TRUE** or **FALSE** in the *`enable`* parameter.

If this stream should be seeded with settings from an already established crypto enabled stream, pass that stream's resource variable in the fourth parameter.

Returns **TRUE** on success, **FALSE** if negotiation has failed or *0* if there isn't enough data and you should try again (only for non-blocking sockets).

Example #22 - stream_socket_enable_crypto() **Example**

```
<?php
$fp = stream_socket_client("tcp://myproto.example.com:31337", $errno,
$errstr, 30);
if (!$fp) {
 die("Unable to connect: $errstr ($errno)");
}
/* Turn on encryption for login phase */
stream_socket_enable_crypto($fp, true, STREAM_CRYPTO_METHOD_SSLv23_CLIENT);
fwrite($fp, "USER god\r\n");
fwrite($fp, "PASS secret\r\n");
/* Turn off encryption for the rest */
```

```
stream_socket_enable_crypto($fp, false);
while ($motd = fgets($fp)) {
 echo $motd;
}
fclose($fp);
?>
```

OpenSSL Functions, and List of Supported Socket Transports

stream_socket_get_name

stream_socket_get_name -- Retrieve the name of the local or remote sockets

Description

string **stream_socket_get_name** (resource `$handle`, bool `$want_peer`)

Returns the local or remote name of a given socket connection. If *`want_peer`* is set to **TRUE** the remote socket name will be returned, if it is set to **FALSE** the local socket name will be returned.

See also stream_socket_accept().

stream_socket_pair

stream_socket_pair -- Creates a pair of connected, indistinguishable socket streams

Description

array **stream_socket_pair** (int `$domain`, int `$type`, int `$protocol`)

stream_socket_pair() creates a pair of connected, indistinguishable socket streams. This function is commonly used in IPC (Inter-Process Communication).

Parameters

`domain`
: The protocol family to be used: **STREAM_PF_INET**, **STREAM_PF_INET6** or **STREAM_PF_UNIX**

`type`
: The type of communication to be used: **STREAM_SOCK_DGRAM**, **STREAM_SOCK_RAW**, **STREAM_SOCK_RDM**, **STREAM_SOCK_SEQPACKET** or **STREAM_SOCK_STREAM**

`protocol`
: The protocol to be used: **STREAM_IPPROTO_ICMP**, **STREAM_IPPROTO_IP**, **STREAM_IPPROTO_RAW**, **STREAM_IPPROTO_TCP** or **STREAM_IPPROTO_UDP**

Note
Please consult the Streams constant list for further details on each constant.

Return Values

Returns an array with the two socket resources on success, or **FALSE** on failure.

Examples

Example #23 - A stream_socket_pair() example

This example shows the basic usage of stream_socket_pair() in Inter-Process Comunication.

```
<?php
```

```
$sockets = stream_socket_pair(STREAM_PF_UNIX, STREAM_SOCK_STREAM,
STREAM_IPPROTO_IP);
$pid     = pcntl_fork();

if ($pid == -1) {
    die('could not fork');

} else if ($pid) {
    /* parent */
   fclose($sockets[0]);

   fwrite($sockets[1], "child PID: $pid\n");
   echo fgets($sockets[1]);

   fclose($sockets[1]);

} else {
   /* child */
   fclose($sockets[1]);

   fwrite($sockets[0], "message from child\n");
   echo fgets($sockets[0]);

   fclose($sockets[0]);
}

?>
```

The above example will output something similar to:

```
child PID: 1378
message from child
```

Notes

Note
This function is not implemented on Windows platforms.

stream_socket_recvfrom

stream_socket_recvfrom -- Receives data from a socket, connected or not

Description

string **stream_socket_recvfrom** (resource `$socket`, int `$length` [, int `$flags` [, string `&$address`]])

The function stream_socket_recvfrom() accepts data from a remote socket up to *`length`* bytes. If *`address`* is provided it will be populated with the address of the remote socket.

The value of *`flags`* can be any combination of the following:

possible values for *`flags`*

STREAM_OOB	Process OOB (out-of-band) data.
STREAM_PEEK	Retrieve data from the socket, but do not consume the buffer. Subsequent calls to fread() or stream_socket_recvfrom() will see the same data.

Example #24 - stream_socket_recvfrom() Example

```
<?php
/* Open a server socket to port 1234 on localhost */
$server = stream_socket_server('tcp://127.0.0.1:1234');

/* Accept a connection */
$socket = stream_socket_accept($server);

/* Grab a packet (1500 is a typical MTU size) of OOB data */
echo "Received Out-Of-Band: '" . stream_socket_recvfrom($socket, 1500,
STREAM_OOB) . "'\n";

/* Take a peek at the normal in-band data, but don't comsume it. */
echo "Data: '" . stream_socket_recvfrom($socket, 1500, STREAM_PEEK) . "'\n";

/* Get the exact same packet again, but remove it from the buffer this time.
*/
echo "Data: '" . stream_socket_recvfrom($socket, 1500) . "'\n";

/* Close it up */
fclose($socket);
fclose($server);
?>
```

Note

If a message received is longer than the *length* parameter, excess bytes may be discarded depending on the type of socket the message is received from (such as UDP).

Note

Calls to stream_socket_recvfrom() on socket-based streams, after calls to buffer-based stream functions (like fread() or stream_get_line()) read data directly from the socket and bypass the stream buffer.

See also stream_socket_sendto(), stream_socket_client(), and stream_socket_server().

stream_socket_sendto

stream_socket_sendto -- Sends a message to a socket, whether it is connected or not

Description

int **stream_socket_sendto** (resource `$socket`, string `$data` [, int `$flags` [, string `$address`]])

The function stream_socket_sendto() sends the data specified by *`data`* through the socket specified by *`socket`*. The address specified when the socket stream was created will be used unless an alternate address is specified in *`address`*.

The value of *`flags`* can be any combination of the following:

possible values for *`flags`*

STREAM_OOB	Process OOB (out-of-band) data.

Example #25 - stream_socket_sendto() Example

```
<?php
/* Open a socket to port 1234 on localhost */
$socket = stream_socket_client('tcp://127.0.0.1:1234');

/* Send ordinary data via ordinary channels. */
fwrite($socket, "Normal data transmit.");

/* Send more data out of band. */
stream_socket_sendto($socket, "Out of Band data.", STREAM_OOB);

/* Close it up */
fclose($socket);
?>
```

See also stream_socket_recvfrom(), stream_socket_client(), and stream_socket_server().

stream_socket_server

stream_socket_server -- Create an Internet or Unix domain server socket

Description

resource **stream_socket_server** (string `$local_socket` [, int `&$errno` [, string `&$errstr` [, int `$flags` [, resource `$context`]]]])

Creates a stream or datagram socket on the specified *`local_socket`*. The type of socket created is determined by the transport specified using standard URL formatting: *transport://target*. For Internet Domain sockets (AF_INET) such as TCP and UDP, the *target* portion of the *`remote_socket`* parameter should consist of a hostname or IP address followed by a colon and a port number. For Unix domain sockets, the *`target`* portion should point to the socket file on the filesystem. *`flags`* is a bitmask field which may be set to any combination of socket creation flags. The default value of flags is **STREAM_SERVER_BIND** | **STREAM_SERVER_LISTEN**.

Note
For UDP sockets, you must use **STREAM_SERVER_BIND** as the *`flags`* parameter.

This function only creates a socket, to begin accepting connections use stream_socket_accept().

If the call fails, it will return **FALSE** and if the optional *`errno`* and *`errstr`* arguments are present they will be set to indicate the actual system level error that occurred in the system-level *socket()*, *bind()*, and *listen()* calls. If the value returned in *`errno`* is *0* and the function returned **FALSE**, it is an indication that the error occurred before the *bind()* call. This is most likely due to a problem initializing the socket. Note that the *`errno`* and *`errstr`* arguments will always be passed by reference.

Depending on the environment, Unix domain sockets may not be available. A list of available transports can be retrieved using stream_get_transports(). See List of Supported Socket Transports for a list of bulitin transports.

Example #26 - Using TCP server sockets

```
<?php
$socket = stream_socket_server("tcp://0.0.0.0:8000", $errno, $errstr);
if (!$socket) {
 echo "$errstr ($errno)<br />\n";
} else {
 while ($conn = stream_socket_accept($socket)) {
   fwrite($conn, 'The local time is ' . date('n/j/Y g:i a') . "\n");
   fclose($conn);
 }
```

```
 fclose($socket);
}
?>
```

The example below shows how to act as a time server which can respond to time queries as shown in an example on stream_socket_client().

Note

Most systems require root access to create a server socket on a port below 1024.

Example #27 - Using UDP server sockets

```
<?php
$socket = stream_socket_server("udp://127.0.0.1:1113", $errno, $errstr,
STREAM_SERVER_BIND);
if (!$socket) {
   die("$errstr ($errno)");
}

do {
   $pkt = stream_socket_recvfrom($socket, 1, 0, $peer);
   echo "$peer\n";
   stream_socket_sendto($socket, date("D M j H:i:s Y\r\n"), 0, $peer);
} while ($pkt !== false);

?>
```

Note

When specifying a numerical IPv6 address (e.g. *fe80::1*), you must enclose the IP in square brackets?for example, *tcp:///[fe80::1]:80*.

See also stream_socket_client(), stream_set_blocking(), stream_set_timeout(), fgets(), fgetss(), fwrite(), fclose(), feof(), and the Curl extension.

stream_socket_shutdown

stream_socket_shutdown -- Shutdown a full-duplex connection

Description

bool **stream_socket_shutdown** (resource `$stream`, int `$how`)

Shutdowns (partially or not) a full-duplex connection.

Parameters

`stream`
: An open stream (opened with stream_socket_client(), for example)

`how`
: One of the following constants: **STREAM_SHUT_RD** (disable further receptions), **STREAM_SHUT_WR** (disable further transmissions) or **STREAM_SHUT_RDWR** (disable further receptions and transmissions).

Return Values

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

Examples

Example #28 - A stream_socket_shutdown() example

```
<?php

$server = stream_socket_server('tcp://127.0.0.1:1337');
$client = stream_socket_client('tcp://127.0.0.1:1337');

var_dump(fputs($client, "hello"));

stream_socket_shutdown($client, STREAM_SHUT_WR);
var_dump(fputs($client, "hello")); // doesn't work now

?>
```

The above example will output something similar to:

```
int(5)

Notice: fputs(): send of 5 bytes failed with errno=32 Broken pipe in
test.php on line 9
int(0)
```

See Also

- fclose()

stream_wrapper_register

stream_wrapper_register -- Register a URL wrapper implemented as a PHP class

Description

bool **stream_wrapper_register** (string `$protocol`, string `$classname`)

stream_wrapper_register() allows you to implement your own protocol handlers and streams for use with all the other filesystem functions (such as fopen(), fread() etc.).

To implement a wrapper, you need to define a class with a number of member functions, as defined below. When someone fopens your stream, PHP will create an instance of *`classname`* and then call methods on that instance. You must implement the methods exactly as described below - doing otherwise will lead to undefined behaviour.

Note
As of PHP 5.0.0 the instance of *`classname`* will be populated with a *`context`* property referencing a *Context Resource* which may be accessed with stream_context_get_options(). If no context was passed to the stream creation function, *`context`* will be set to **NULL**.

stream_wrapper_register() will return **FALSE** if the *`protocol`* already has a handler.

bool **stream_open** (string `$path`, string `$mode`, int `$options`, string `$opened_path`)

This method is called immediately after your stream object is created. *`path`* specifies the URL that was passed to fopen() and that this object is expected to retrieve. You can use parse_url() to break it apart.

`mode` is the mode used to open the file, as detailed for fopen(). You are responsible for checking that *`mode`* is valid for the *`path`* requested.

`options` holds additional flags set by the streams API. It can hold one or more of the following values OR'd together.

Flag	Description
STREAM_USE_PATH	If *`path`* is relative, search for the resource using the include_path.
STREAM_REPORT_ERRORS	If this flag is set, you are responsible for raising errors using trigger_error() during opening of the stream. If this flag is not set, you should not raise any errors.

If the *path* is opened successfully, and STREAM_USE_PATH is set in *options*, you should set *opened_path* to the full path of the file/resource that was actually opened.

If the requested resource was opened successfully, you should return **TRUE**, otherwise you should return **FALSE**

void **stream_close** (void)

This method is called when the stream is closed, using fclose(). You must release any resources that were locked or allocated by the stream.

string **stream_read** (int $count)

This method is called in response to fread() and fgets() calls on the stream. You must return up-to *count* bytes of data from the current read/write position as a string. If there are less than *count* bytes available, return as many as are available. If no more data is available, return either **FALSE** or an empty string. You must also update the read/write position of the stream by the number of bytes that were successfully read.

int **stream_write** (string $data)

This method is called in response to fwrite() calls on the stream. You should store *data* into the underlying storage used by your stream. If there is not enough room, try to store as many bytes as possible. You should return the number of bytes that were successfully stored in the stream, or 0 if none could be stored. You must also update the read/write position of the stream by the number of bytes that were successfully written.

bool **stream_eof** (void)

This method is called in response to feof() calls on the stream. You should return **TRUE** if the read/write position is at the end of the stream and if no more data is available to be read, or **FALSE** otherwise.

int **stream_tell** (void)

This method is called in response to ftell() calls on the stream. You should return the current read/write position of the stream.

bool **stream_seek** (int $offset, int $whence)

This method is called in response to fseek() calls on the stream. You should update the read/write position of the stream according to *offset* and *whence*. See fseek() for more information about these parameters. Return **TRUE** if the position was updated, **FALSE** otherwise.

bool **stream_flush** (void)

This method is called in response to fflush() calls on the stream. If you have cached data in your stream but not yet stored it into the underlying storage, you should do so now. Return **TRUE** if the cached data was successfully stored (or if there was no data to store), or **FALSE** if the data could not be stored.

array **stream_stat** (void)

This method is called in response to fstat() calls on the stream and should return an array containing the same values as appropriate for the stream.

bool **unlink** (string `$path`)

This method is called in response to unlink() calls on URL paths associated with the wrapper and should attempt to delete the item specified by *`path`*. It should return **TRUE** on success or **FALSE** on failure. In order for the appropriate error message to be returned, do not define this method if your wrapper does not support unlinking.

Note
Userspace wrapper unlink method is not supported prior to PHP 5.0.0.

bool **rename** (string `$path_from`, string `$path_to`)

This method is called in response to rename() calls on URL paths associated with the wrapper and should attempt to rename the item specified by *`path_from`* to the specification given by *`path_to`*. It should return **TRUE** on success or **FALSE** on failure. In order for the appropriate error message to be returned, do not define this method if your wrapper does not support renaming.

Note
Userspace wrapper rename method is not supported prior to PHP 5.0.0.

bool **mkdir** (string `$path`, int `$mode`, int `$options`)

This method is called in response to mkdir() calls on URL paths associated with the wrapper and should attempt to create the directory specified by *`path`*. It should return **TRUE** on success or **FALSE** on failure. In order for the appropriate error message to be returned, do not define this method if your wrapper does not support creating directories. Posible values for *`options`* include **STREAM_REPORT_ERRORS** and **STREAM_MKDIR_RECURSIVE**.

Note
Userspace wrapper mkdir method is not supported prior to PHP 5.0.0.

bool **rmdir** (string `$path`, int `$options`)

This method is called in response to rmdir() calls on URL paths associated with the wrapper and should attempt to remove the directory specified by *`path`*. It should return **TRUE** on success or **FALSE** on failure. In order for the appropriate error message to be

returned, do not define this method if your wrapper does not support removing directories. Possible values for *`options`* include **STREAM_REPORT_ERRORS**.

Note
Userspace wrapper rmdir method is not supported prior to PHP 5.0.0.

bool **dir_opendir** (string `$path`, int `$options`)

This method is called immediately when your stream object is created for examining directory contents with opendir(). *`path`* specifies the URL that was passed to opendir() and that this object is expected to explore. You can use parse_url() to break it apart.

array **url_stat** (string `$path`, int `$flags`)

This method is called in response to stat() calls on the URL paths associated with the wrapper and should return as many elements in common with the system function as possible. Unknown or unavailable values should be set to a rational value (usually **0**).

`flags` holds additional flags set by the streams API. It can hold one or more of the following values OR'd together.

Flag	**Description**
STREAM_URL_STAT_LINK	For resources with the ability to link to other resource (such as an HTTP Location: forward, or a filesystem symlink). This flag specified that only information about the link itself should be returned, not the resource pointed to by the link. This flag is set in response to calls to lstat(), is_link(), or filetype().
STREAM_URL_STAT_QUIET	If this flag is set, your wrapper should not raise any errors. If this flag is not set, you are responsible for reporting errors using the trigger_error() function during stating of the path.

string **dir_readdir** (void)

This method is called in response to readdir() and should return a string representing the next filename in the location opened by **dir_opendir()**.

bool **dir_rewinddir** (void)

This method is called in response to rewinddir() and should reset the output generated by **dir_readdir()**. i.e.: The next call to **dir_readdir()** should return the first entry in the location

returned by **dir_opendir()**.

bool **dir_closedir** (void)

This method is called in response to closedir(). You should release any resources which were locked or allocated during the opening and use of the directory stream.

The example below implements a var:// protocol handler that allows read/write access to a named global variable using standard filesystem stream functions such as fread(). The var:// protocol implemented below, given the URL "var://foo" will read/write data to/from $GLOBALS["foo"].

Example #29 - A Stream for reading/writing global variables

```
<?php

class VariableStream {
   var $position;
   var $varname;

   function stream_open($path, $mode, $options, &$opened_path)
   {
       $url = parse_url($path);
       $this->varname = $url["host"];
       $this->position = 0;

       return true;
   }

   function stream_read($count)
   {
       $ret = substr($GLOBALS[$this->varname], $this->position, $count);
       $this->position += strlen($ret);
       return $ret;
   }

   function stream_write($data)
   {
       $left = substr($GLOBALS[$this->varname], 0, $this->position);
       $right = substr($GLOBALS[$this->varname], $this->position +
strlen($data));
       $GLOBALS[$this->varname] = $left . $data . $right;
       $this->position += strlen($data);
       return strlen($data);
   }

   function stream_tell()
   {
       return $this->position;
   }

   function stream_eof()
   {
       return $this->position >= strlen($GLOBALS[$this->varname]);
   }

   function stream_seek($offset, $whence)
   {
```

```
        switch ($whence) {
            case SEEK_SET:
                if ($offset < strlen($GLOBALS[$this->varname]) && $offset >=
0) {
                     $this->position = $offset;
                     return true;
                } else {
                     return false;
                }
                break;

            case SEEK_CUR:
                if ($offset >= 0) {
                     $this->position += $offset;
                     return true;
                } else {
                     return false;
                }
                break;

            case SEEK_END:
                if (strlen($GLOBALS[$this->varname]) + $offset >= 0) {
                     $this->position = strlen($GLOBALS[$this->varname]) +
$offset;
                     return true;
                } else {
                     return false;
                }
                break;

            default:
                return false;
        }
    }
}

stream_wrapper_register("var", "VariableStream")
   or die("Failed to register protocol");

$myvar = "";

$fp = fopen("var://myvar", "r+");

fwrite($fp, "line1\n");
fwrite($fp, "line2\n");
fwrite($fp, "line3\n");

rewind($fp);
while (!feof($fp)) {
   echo fgets($fp);
}
fclose($fp);
var_dump($myvar);

?>
```

stream_wrapper_restore

stream_wrapper_restore -- Restores a previously unregistered built-in wrapper

Description

bool **stream_wrapper_restore** (string `$protocol`)

Restores a built-in wrapper previously unregistered with stream_wrapper_unregister().

Parameters

`protocol`

Return Values

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

stream_wrapper_unregister

stream_wrapper_unregister -- Unregister a URL wrapper

Description

bool **stream_wrapper_unregister** (string `$protocol`)

Allows you to disable an already defined stream wrapper. Once the wrapper has been disabled you may override it with a user-defined wrapper using stream_wrapper_register() or reenable it later on with stream_wrapper_restore().

Parameters

`protocol`

Return Values

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