psArchive®

User Manual

version 0.1.2

Table of Contents

1) Introduction

To begin with, psArchive is a PHP 5 extension designed to provide an easy-to-use interface for manipulating all the most common archive files used in web projects. It is especially designed for web applications that need to understand, to produce or even edit archives in order to function.

The applications of this extension can be of a wide variety: one could generate periodic archives of logs, reports or documents, multiple files can be clustered together for easy transmittal through the Internet etc.

It is true that on almost all desktop computers today one can find at least one kind of compression software, however, when it comes to the fast growing PHP developer community all there is to work with are a few scripts that are slow (due to parsing archives in pure PHP), limited (they can handle only small workloads safely and none of the can properly edit an archive), rigid (unable to configure themselves to optimize resource management for heavy solicited servers) and many times even messy (memory management is left almost entirely on PHP's shoulders). Well, psArchive is a **very** affordable way to change all that... FREE!!!

A) First of all, it's a PHP extension written in C which means a number of things:

- it's fast and stable even at high workloads;
- it eliminates the overhead of loading/including heavy PHP scripts;
- memory management is properly handled keeping resource loss at a minimum;
- it makes your code a **lot** cleaner, easier to write and to understand.

B) Secondly, this extension has all formats based on embedded code, thus **not** being dependent on **any** other external library, therefore you don't need to install other packages/extensions in order to make use of it, unlike the PHP scripts mentioned earlier.

C) psArchive currently supports the following formats: zip, gz, bz, bz2, tar, tar.gz, tar.bz, tar.bz2, tgz, tbz, tbz2; also psArchive is sustained by a team of developers that is ready to implement support for other archive formats on demand (with no extra fees).

D) psArchive supports *standard ZIP encryption/decryption* and even though this is not a recommended way of protecting very important data due to it's weak algorithm this feature may prove to be extremely useful in situations in which you want either to encrypt an archive in a moderately safe manner or simply that you wish to open a password protected file.

E) psArchive can be configured to store data either on hard disk (for extreme memory preservation), in compressed memory (for a faster, low memory consumption way of handling files when heavily multi-threaded) or in raw memory (for extreme speed of processing), leaving the developer to decide which configuration option is best for the target application.

F) psArchive is **extremely** easy to use, given it's Object Oriented architecture and clear methods for manipulation one can practically start working with it as soon as he/she sees a list of what those methods and functions are.

G) psArchive is designed to avoid creating the uncompressed archive into memory before writing to disk so that precious RAM is not wasted pointlessly (this is especially true with tar archives).

H) Many other things can be said about psArchive, for example that it comes along with two extremely fast functions for data compression/decompression to be used in trivial situations.

In conclusion psArchive is an affordable, high performance PHP extension optimized for server-side archive generation/editing/extraction etc.

2) Features and requirements

Features:

- **Supports archive *editing*! You can <u>add</u>, <u>remove</u>, <u>rename</u> and <u>copy</u> files, you can <u>change file contents</u>, <u>attributes</u> and <u>modification time</u>, you can <u>create new files</u> and <u>directories</u>, *all* right inside the archive before output.**
- **Supports the following archive formats:**
  1. ZIP (single volumes);
  2. TAR (tarball - tape archive);
  3. GZ (Gzip compressed format);
  4. BZ, BZ2 (Bzip/Bzip2 compressed format);
  5. TAR.GZ, TGZ (Gzipped Tarball);
  6. TAR.BZ2, TAR.BZ, TBZ2, TBZ (Bzipped Tarball);
  7. Other formats can be supported on demand, such as RAR or ACE support (since these formats are uncommon in web environments we decided not to include them in order to keep the extension size as small as possible);
- **Supports both reading from and writing to password encrypted ZIP files using standard ZIP encryption;**
- **Configurable buffer storage for optimized and threadsafe memory handling:**
  1. In raw memory (for optimized speed);
  2. In compressed memory (normal operation);
  3. On disk (for optimized memory consumption);
- **Supports wildcards for both reading files from disk as well as filtering the archive**

contents. Basically, wherever there is a possibility that wildcards could be used, they actually can be!;

- **Provides 2 functions for extremely fast in-memory data compression-decompression using LZF algorithm. These functions correctly compress/decompress data blocks far greater than 1024 bytes;**
- **An advanced as well as uniform archive manipulation design which enables you to make all possible alterations to an archive (create, load, file manipulation, extractions etc.) and then simply output the archive in all the desired formats without re-reading files, or re-altering them;**
- **Supports object importing! Basically if you have two or more opened archives in your script you can import files from one archive object into another directly. Also you can filter the importing by using wildcards;**
- **Fully embedded! Needs no external libraries in order to function. All in just over 400 KB;**
- **Object Oriented Architecture! This library provides a total of two functions (the LZF compress/decompress) and one class. It is extremely easy to use!**

Requirements:

- **PHP 5**;
- **A Linux-based web server** (this version of **psArchive** is only compiled for Linux-based platforms. We do not recommend the use of psArchive on a Windows server since file security is weaker and much of the security is left to the script to handle. HOWEVER! If you really need a Windows compiled version of this extensions contact us at support@palos.ro and we shall provide you with both a DLL as well as installation instructions at no extra fees), **Apache is recommended**;

3) Installing psArchive

This is the installation procedure for a Linux web-server (in case you need Windows support please contact support@palos.ro).

First of all decompress the file archive containing psArchive:

For the tar.gz version do:

```
tar -zxvf psArchive-X.tar.gz
```

For the tar.bz2 version do:

```
tar -jxvf psArchive-X.tar.bz2
```

where X represents the version/revision numbers.

There are two ways to install psArchive. One is to use the provided binary ‘psArchive.so’. If this works for you then this is the easier way. If not, simply compile psArchive.

To compile psArchive one would need to call (in this order):

```
./__clean

./__init

./__make
```

For everything to work fine you must have the 'phpize' executable an visible system-wide. All should work just fine if you use GCC versions prior to 4.0. If not, you must force the compiler to use GCC 3.x by issuing the following command at the command line before the ones above:

```
export CC=gcc32
```

An then continue with './__clean', etc.

Afterwards, installing the PHP extension is a walk in the park, and for that you only need the following:

1. psArchive.so	This file is located in the subdirectory modules. Put the extension into the PHP extension modules directory (commonly known as `/usr/lib/php/modules` but may differ). If you do not know what this path is you may check the `extension_dir` directive in your php.ini;
2. psArchive.ini	Put the ini file for loading the extension and configuring it inside the `/etc/php.d` directory. This is the extension loading and configuration directory for PHP 5 (it is possible to be a different path in your server's case).
3. A web server restart.	Usually you can do this with the command: `/etc/init.d/httpd restart`
4. Evrika, you're set! Enjoy!	

4) Configuring psArchive for optimum performance

This is one of the strong features of psArchive. The configuration of psArchive is done by altering the two configuration directives in psArchive.ini (`psArchive.storage` and `psArchive.bufferSize`) as you see fit and after that restarting the webserver. Simple right? Well it usually is, but sometimes the hard part is actually deciding on the settings that might be best for you!

Basically one of the things psArchive does in order to maintain such a high level of maneuverability is to keep the archive files temporarily stored into a "storage bin". This storage bin can be set to exist into three different environments. So here are the options:

Option `psArchive.storage = x`

Where x is as follows...

`0 -` This option means that the files are stored in pure RAM! While this option is the fastest way to go, it consumes a lot of memory (basically you must have enough space in ram to hold the complete, decompressed archives for all running threads). This option is recommended only if you have a lot of ram and if you absolutely need extreme speed.

1 `-` This option means that the files are stored in compressed RAM! Basically it is the same as option 0 but with a twist. The files are first compressed before putting into RAM using the speedy LZF functions. This provides a dramatic decrease of memory consumption but with a slight cost in speed. This is the default operation mode.

`2 -` This option means that the files are stored on disk thus preserving as much RAM as possible! With this option you only need enough free RAM to hold the largest file in the archive in uncompressed state. All archive objects created in one script run are stored into just one file which is deleted at the end of the script execution. Also when deleting files from archives the empty spaces created in the storage bin are then reused. These tactics are used in order to minimize fragmentation in heavy workload scenarios. This option should usually be slower than option 1 but interestingly enough, using this option might even provide faster times when (and **only** when) not running in heavy multi threaded situations. Why?, Simple, because the overhead of compressing/decompressing a file from memory might be slightly greater then the overhead of reading/writing it from disk (especially if you have a high speed HDD drive). However, in multi threaded environments (or if the HDD is under heavy workload) this option **will** prove to be a bit slower than option 1. This option is used when dealing with very large archives.

So, is it clear? If not, just follow these recommendations:

- **If unsure what to do just leave option 1 selected;**
- **If dealing with a highly multi threaded script handling archive at most a few (2~5) MegaBytes in size then use option 0;**
- **If dealing with huge archive (tens or hundreds of megabytes) use option 2;**

NOTE! A word of caution when dealing with very large archive files. Web browsers usually have a timeout (for example 120 seconds) after which, if the script did not successfully provide some kind of output, they simply reissue the HTTP request. This causes the web server to start a new thread of the exactly same script with the exact same parameters before the initial thread finished. This will in turn put a huge stress on you server! To avoid this try to output and flush something to the web browser before starting to process the large archive/file. This will usually convince the browser that there is no need to reissue the request since the server is processing the script. This is how you do this:

```
...
echo “Starting to process the very large archive file...”;
flush();
....
```

Option `psArchive.bufferSize = x`

Where x is a number in bytes specifying the buffer size used for reading to and writing from a file on disk. Basically, when psArchive reads/writes a file it does not request or send the whole contents. It splits the data into smaller chunks and reads/writes them sequentially. This is a blessing for slower servers that are heavily multi threaded! In these cases you should decrease this value to get a improve the server's processing. However, usually you should leave this setting to the default (1048576 – meaning exactly 1 MegaByte). If you really want extreme speed you might increase this value but be careful, since you might be walking on thin ice.

Option `psArchive.temporaryDir = "xxx"`

Where xxx is a path on disk where you want psArchive to store temporary files. You could set this to point to a directory located on a disk or partition that is especially built for such tasks.

5) Function and class reference

`string` **`psACompressData`**`( string` **`$prData`** `)`

This function takes the data contained in the string $prData and returns it's compressed form. It uses the high speed LZF algorithm and provides a slightly lower compression ratio than gzip level 1. It can support large data blocks. In some extreme cases (such as very small data) it can provide an output that is 1-byte larger than the input.

Parameters:

$prData – data string to compress

Returns:

string – compressed data

`string` **`psAUncompressData`**`( string` **`$prData`** `)`

This function takes the data string $prData previously compressed with psACompress() and returns it decompressed.

Parameters:

$prData – data string to decompress

Returns:

string – decompressed data

`class` **`psArchive`**

This class contains all methods needed to manipulate an archive.

`int psArchive::`**`getCaps`**`( string` **`$prFormat`** `)`

This method returns psArchive's capability to handle the specified format.

Parameters:

$prFormat – specifies the format to inquire about. It must be an extension without the preceding dot (i.e. 'zip', 'tar.gz', 'tbz' etc.) and all letters must be lowercase.

Returns:

Returns one of the following constant integer values (defined by psArchive):

PSARCHIVE_CAN_READ = meaning that the archive format can be read

PSARCHIVE_CAN_WRITE = meaning that the archive format can be generated

PSARCHIVE_CAN_RW = PSARCHIVE_CAN_READ + PSARCHIVE_CAN_WRITE

Note: this method can be called statically without the need to instantiate an object. Example:

var_dump(psArchive::getCaps('tgz'));

`string psArchive::`**`getMime`**`( string `**`$prFormat`**` )`

This method returns the mime type corresponding to the specified format. For example if you want to output the archive directly to the browser for instant download you must call the php header function prior to calling the psArchive::toBrowser() method telling the browser what kind of file will be transmitted. You do this like this:

```
header('Content-type: xxxxxxx ;');
```

Where xxxxxxx is the mime type specifying the file type (something similar to **image/jpeg** or **text/html**). That is where this function comes in by returning the exact mime type needed to transmit the given format through the web or through mail.

Parameters:

$prFormat – specifies the format to inquire about. It must be an extension without the preceding dot (i.e. 'zip', 'tar.gz', 'tbz' etc.) and all letters must be lowercase.

Returns:

string – mime type

Note: this method can be called statically without the need to instantiate an object. Example:

var_dump(psArchive::getMime('tgz'));

`boolean psArchive::`**`isEmpty`**`( )`

Checks to see if the archive object is empty.

Returns:

TRUE – if archive object is empty

FALSE – if archive object is not empty

`boolean psArchive::`**`check`**`( string $prPath [, string $prFormat] )`

Checks to see if an existing file is a valid archive. Note that this function generates an error if used to check files of different formats than those supported by psArchive.

Parameters:

$prPath – the file given for checking.

$prFormat – manually specify the file format in case it has a different extension. If this parameter is not specified then the format is guessed from the file extension.

Returns:

TRUE – file is a valid archive

FALSE – file is not a valid archive

```
boolean psArchive::load( string $prPath
                            [, string $prFormat
                            [, boolean $prOverwrite
                            [, string $prPassword]]] )
```

Loads the contents(files and directories) of an existing archive file into the current archive object. If file is invalid an error is generated.

Parameters:

$prPath – the file given for loading.

$prFormat – manually specify the file format in case it has a different extension. If this parameter is not specified then the format is guessed from the file extension.

$prOverwrite – specifies if to overwrite duplicate entries found or not; this is true by default

$prPassword – the password needed to open the archive (currently it is ignored if the file is not a ZIP file)

Returns:

TRUE – if archive was successfully loaded into memory

FALSE – if no file was loaded or an error occurred

```
boolean psArchive::add( string $prPath
                            [, boolean $prOverwrite
                            [, boolean $prRecursive]] )
```

Adds a number of files and directories to the archive object.

Parameters:

$prPath – the path to search for file addition; **wildcards can be used** (i.e. “foo*/foobar.?xt”)!

$prOverwrite – specifies if to overwrite duplicate entries found or not; this is true by default

$prRecursive – specifies if directories are to be added in depth (including contained files and directories) or not; this is true by default

Returns:

TRUE – if files were successfully loaded into memory

FALSE – if no file was added or an error occurred

```
boolean psArchive::import( object $prArchive, string $prPath
                           [, boolean $prOverwrite
                           [, boolean $prRecursive]] )
```

Imports files directly from another psArchive object into the current one.

Parameters:

$prArchive – the psArchive object to import from.

$prPath – the path to search for file import; **wildcards can be used** (i.e. “foo*/foobar.?xt”)!

$prOverwrite – specifies if to overwrite duplicate entries found or not; this is true by default

$prRecursive – specifies if directories are to be imported along with their contents or not; this is true by default

Returns:

TRUE – if files were successfully imported

FALSE – if no file was imported or an error occurred

```
boolean psArchive::newFile( string $prPath
                            [, string $prData] )
```

Creates a new file into the archive.

Parameters:

$prPath – file name to use

$prData – the data to put into the file; if not specified the file will be empty

Returns:

TRUE – if the file was successfully created

FALSE – if the name already exists or an error occurred

```
boolean psArchive::newDirectory( string $prPath )
```

Creates a new directory entry into the archive. You do not need to do this in order to add files into a new directory. For example in an empty archive, to add the file “foo/foobar.txt” you only need to call newFile(‘foo/foobar.txt’,’foo foo foo’) or any other file addition command for that matter.

Parameters:

$prPath – directory name to use

Returns:

TRUE – if the directory entry was successfully created

FALSE – if the name already exists or an error occurred

```
boolean psArchive::extract( string $prPath, string $prOutPath
                            [, boolean $prRecursive] )
```

Extracts files from the archive to the disk.

__**Parameters**__**:**

$prPath – the path to search for extraction; __**wildcards can be used**__ (i.e. “foo*/foobar.?xt”)!

$prOutPath – specifies the path in which to extract the files

$prRecursive – specifies if directories are to be extracted along with their contents or not; this is true by default

__**Returns**__:

TRUE – if files were successfully extracted

FALSE – if an error occurred

```
boolean psArchive::extractAll( string $prOutPath )
```

Extracts all files and directories from the archive to the disk.

__**Parameters**__**:**

$prOutPath – specifies the path in which to extract the files

__**Returns**__:

TRUE – if files were successfully extracted

FALSE – if an error occurred

```
boolean psArchive::remove( string $prPath
                            [, boolean $prRecursive] )
```

Removes files from the archive.

__**Parameters**__**:**

$prPath – the path to search for extraction; __**wildcards can be used**__ (i.e. “foo*/foobar.?xt”)!

$prRecursive – specifies if directories are to be removed along with their contents or not; this is true by default.

The archive file names are memorized with their full path as a list (not as a tree like on the HDD) therefore a directory entry may be erased but it’s files can remain afterwards. If you need further clarification please contact support@palos.ro.

__**Returns**__:

TRUE – if files were successfully removed

FALSE – if an error occurred

```
boolean psArchive::removeAll( )
```

Removes all files and directories from the archive leaving the archive totally empty.

Returns:
TRUE – if files were successfully removed
FALSE – if an error occurred

```
array psArchive::filter( string $prPath
                         [, boolean $prRecursive] )
```

Filters the file list of the current archive and returns the matched files as an array. If you want all files to be listed just pass '*' as $prPath.

Parameters:
$prPath – the path to search for matching; **wildcards can be used** (i.e. "foo*/foobar.?xt")!
$prRecursive – specifies if directories are to be listed along with their contents or not; this is true by default; note that this does not mean that the search pattern will be applied to all files in the structure (like the shell would do). For example if you have the following files in the archive:

```
/foo.bar
/foo.bar/foobar.txt
/foo.bar/dummy.txt
```

The command $object->filter('*.bar') would return all three files because the $prRecursive parameter tells the function to return the entire contents of the directories matched (from the beginning of the path), in this case `/foo.bar`.

Returns:
array() – list of files from the archive that matched the search pattern
FALSE – if an error occurred

```
boolean psArchive::copy( string $prPath, string $prNewPath
                         [, boolean $prRecursive] )
```

Copies a file or directory under a new name in the archive.

Parameters:
$prPath – the existing file or directory name
$prNewPath – the new file or directory name
$prRecursive – specifies if directories are to be copied along with their contents or not; this is true by default

Returns:
TRUE – if files were copied successfully
FALSE – if an error occurred

```
boolean psArchive::rename( string $prPath, string $prNewPath
                            [, boolean $prRecursive] )
```

Renames (or moves) a file or directory to a new path in the archive.

Parameters:

$prPath – the existing file or directory name

$prNewPath – the new file or directory name

$prRecursive – specifies if directories contents should be renamed also or not; this is true by default

If renaming a directory that contains other files and/or directories you must know that the $prRecursive parameter will determine if all files and directories under the given directory will be renamed too. This is because the files are stored into memory as a list of absolute paths rather than as a tree. So you can rename a directory entry but all the files and directories it used to contain mai remain untouched. For example:

If at one time the archive contains the following:

`/foo` – directory
`/foo/foobar.txt` - file

Let's say you want to rename directory /foo with /blog. If you specify false as the $prRecursive parameter then the archive will contain the following:

`/blog` – directory
`/foo/foobar.txt` - file

However if you specify true as the $prRecursive parameter then the archive will contain the following:

`/blog` – directory
`/blog/foobar.txt` - file

If you need further clarification please contact support@palos.ro.

Returns:

TRUE – if files were renamed successfully

FALSE – if an error occurred

```
boolean psArchive::isEntry( string $prPath)
```

Checks if the specified entry (file or directory) exists in the archive. Note that this has to be a distinct entry, meaning that if you check for 'foo' and all you have in the archive is the file 'foo/foobar.txt' the function will return false.

Parameters:

$prPath – the file name to search for

Returns:

TRUE – if entry exists

FALSE – if entry does not exist

`boolean psArchive::`**`isFile`**`( string $prPath)`

Checks if the specified entry exists in the archive and if it is a file.

Parameters:
$prPath – the file name to search for

Returns:
TRUE – if entry exists and is a file
FALSE – if entry does not exist or is a directory

`boolean psArchive::`**`isDirectory`**`( string $prPath)`

Checks if the specified entry exists in the archive and if it is a directory. Note that this has to be a distinct entry, meaning that if you check for ‘foo’ and all you have in the archive is the file ‘foo/foobar.txt’ the function will return false.

Parameters:
$prPath – the directory name to search for

Returns:
TRUE – if entry exists and is a directory
FALSE – if entry does not exist or is a file

`int psArchive::`**`getType`**`( string `**`$prPath`**` )`

Returns the type of the specified entry (directory or file).

Parameters:
$prPath – the file name to search for

Returns:
Returns one of the following constant integer values (defined by psArchive):

PSARCHIVE_FILE = meaning that the entry is a file

PSARCHIVE_DIR = meaning that the entry is a directory

FALSE – if entry does not exist

`int psArchive::`**`getAttributes`**`( string `**`$prPath`**` )`

Returns the permission attributes of the specifies entry (file or directory) in Unix octal format as a string (i.e. ‘0644’).

Parameters:
$prPath – the file name to search for

Returns:
string – the file permission attributes in Unix octal format

`boolean psArchive::`**`setAttributes`**`( string `**`$prPath,`**` string `**`$prAttr`**` )`

Sets the permission attributes for the given entry (file or directory).

Parameters:
$prPath – the file name to search for
$prAttr – the permission attributes to apply

Returns:
TRUE – if the attributes were applied successfully
FALSE – if the entry does not exist

`int psArchive::`**`getMTime`**`( string `**`$prPath`**` )`

Returns the last modification time of the given entry as a Unix timestamp.

Parameters:
$prPath – the file name to search for

Returns:
int – the file last modification time as a Unix timestamp

`boolean psArchive::`**`setMTime`**`( string `**`$prPath,`**` string `**`$prMTime`**` )`

Sets the last modification time for the given entry.

Parameters:
$prPath – the file name to search for
$prMTime – last modification time as a Unix timestamp.

Returns:
TRUE – if the time was applied successfully
FALSE – if the entry does not exist

```
int psArchive::getFileSize( string $prPath )
```

Returns the size of a file inside the archive.

Parameters:
$prPath – the file name to search for

Returns:
int – the file size
FALSE – if file does not exist or is a directory

```
string psArchive::getFileContents( string $prPath )
```

Returns the contents of a file inside the archive.

Parameters:
$prPath – the file name to search for

Returns:
string – the file contents
FALSE – if file does not exist or is a sirectory

```
boolean psArchive::putFileContents( string $prPath, string $prData )
```

Changes the contents for a file inside the archive.

Parameters:
$prPath – the file name to search for
$prData – new data to put inside the specified file.

Returns:
TRUE – if the data was changed successfully
FALSE – if the file does not exist or is a directory

```
boolean psArchive::toFile(string $prPath
                            [, string $prFormat
                            [, string $prPassword]] )
```

Generates the archive file and saves it under the name $prPath.

<u>Parameters</u>:

$prPath – the file name of the generated archive

$prFormat – the archive file format to use for generation; if not specified it will be guessed from the file extension. This must be an extension (i.e. 'zip') and all letters must be lowercase.

$prPassword – the password to encrypt the file with; this is currently ignored unless the archive is ZIP.

<u>Returns</u>:

TRUE – if the archive was generated successfully

FALSE – if an error occurred

```
boolean psArchive::toBrowser( string $prFormat
                              [, string $prPassword] )
```

Generates the archive file and sends it directly to the browser for download.

<u>Parameters</u>:

$prFormat – the archive file format to use for generation. This must be an extension (i.e. 'zip') and all letters must be lowercase.

$prPassword – the password to encrypt the file with; this is currently ignored unless the archive is ZIP.

<u>Returns</u>:

TRUE – if the archive was generated successfully

FALSE – if an error occurred

```
string psArchive::toBuffer( string $prFormat
                            [, string $prPassword] )
```

Generates the archive file and returns it as a string.

<u>Parameters</u>:

$prFormat – the archive file format to use for generation. This must be an extension (i.e. 'zip') and all letters must be lowercase.

$prPassword – the password to encrypt the file with; this is currently ignored unless the archive is ZIP.

<u>Returns</u>:

string – the generated archive file

FALSE – if an error occurred

6) Example scripts

1) Creating an archive on-the-fly...

```
$arc =& new psArchive();
$arc->newFile('foo/foobar.txt',
            'The little brown fox jumped over the lazy dog.');
$arc->toFile('/tmp/testing.zip',false,'superman');
```

2) Loading an archive and viewing it's contents...

```
$arc =& new psArchive();
$arc->load('foo.tar.gz');
// var_dump is a php construct that shows the contents of any variable

// in the browser, but of course, you knew that... :)
var_dump( $arc->filter("*") );
```

3) Extracting all the JPG files from an archive into the directory foo/output...

```
$arc =& new psArchive();
$arc->load('images.tgz');
$arc->extract('*.jpg','foo/output');
```

4) Importing a directory from another pArchive object...

```
$arc =& new psArchive();
$arc->load('images.tgz');
$arc2 =& new psArchive();
$arc2->load('sounds.tgz');

$arc->import($arc2,'ballads/*');
// show me what happened
var_dump( $arc->filter("*") );
```

5) Loading an archive file of a different extension...

```
$arc =& new psArchive();
// odt is the text document extension of OpenOffice but it is actually

// a zip archive containing the files needed for the document
$arc->load('My Big Bocument.odt','zip');
// show me the contents
var_dump( $arc->filter("*") );
```

6) Single-file archives...

```
$arc =& new psArchive();
$arc->newFile('foo/foobar.txt','This is the file contents!');
// if the archive only has one file we can save it in simple gz format
$arc->toFile('singlef.gz');
// or bz2 format
$arc->toFile('singlef.bz2');
```

7) Editing an archive an generating it in multiple formats...

```
$arc =& new psArchive();
$arc->load('images.tgz');

// set a file's modification time to now
$arc->setMTime('image_logo.jpg',time());
// change the file's name
$arc->rename('image_logo.jpg','big_company_logo.JPG');
// duplicate the file under a new name
$arc->copy('big_company_logo.JPG','same_big_company_logo.JPG');

$arc->toFile('first_try.tbz');
$arc->toFile('bomberman.tar.gz');
$arc->toFile('encrypted.zip');
$arc->toFile('super.tgz');
```

8) More archive strain...

```
$arc =& new psArchive();
// add some files and load some archives
$arc->add('/etc/httpd');
$arc->load('images.tgz');
$arc->load('sounds.tar.bz2');
$arc->load('license.gz');
$arc->add('/var/www/html');

// do some damage
$arc->rename('folk/romanian', 'ethnic_music');
$arc->rename('landscapes/danube', 'nice_pictures');
$arc->rename('etc', 'configuration');

// some more...
$arc->copy('var', 'websites');

// send to browser and download
header('Content-type: '.$arc->getMime('zip'));
header('Content-disposition: attachment; filename="test.zip";');
$arc->toBrowser('zip','mother');
```

7) Contact information

For contact and information about developing with psArchive contact support@palos.ro or valeriu@palos.ro.