Zip Plugin 2.0

Read and write zip archives with Hollywood

Andreas Falkenhahn

Table of Contents

1 General information 1

1.1 Introduction 1
1.2 Terms and conditions 1
1.3 Requirements 2
1.4 Installation 2

2 About zip.hwp 3

2.1 Credits 3
2.2 Frequently asked questions 3
2.3 Known issues 3
2.4 Future 3
2.5 History 3

3 Usage 5

3.1 Using the plugin 5
3.2 Zip archives as directories 5
3.3 Extracting files 6
3.4 Modifying archives 7
3.5 Zip archive basics 8
3.6 Creating zip archives 8
3.7 Linking files 8

4 Function reference 11

4.1 zip.AddDirectory 11
4.2 zip.AddFile 11
4.3 zip.CloseArchive 14
4.4 zip.DeleteFile 15
4.5 zip.ExtractFile 15
4.6 zip.GetFileAtIndex 16
4.7 zip.GetFileAttributes 16
4.8 zip.GetFileComment 18
4.9 zip.GetObjectType 18
4.10 zip.LocateFile 19
4.11 zip.OpenArchive 19
4.12 zip.RenameFile 20
4.13 zip.SetDefaultPassword 21
4.14 zip.SetFileComment 22
4.15 zip.SetFileCompression 22
4.16 zip.SetFileEncryption 23
4.17 zip.SetFileTime 24

Appendix A Licenses 25
A.1 LibZip license 25
A.2 AES encryption support 25

Index 27

1 General information

1.1 Introduction

This plugin allows Hollywood scripts to read and write zip archives. It uses Hollywood's file and directory adapter plugin interfaces which allow you to iterate through zip archives as if they were normal directories. Files within zip archives can also be accessed as if the zip archive was a normal directory. It is not necessary to unpack a file stored in a zip archive to a temporary file before it can be opened. Hollywood's file adapter plugin interface allows direct streaming from the zip archive into the respective file handler.

Additionally, zip.hwp offers a range of functions to read, modify, and write zip archives. New zip archives can be created, existing zip archives can be opened and modified. There is a variety of functions which allows you to read, change, and write attributes of files stored in zip archives. On top of that, zip.hwp also supports reading and storing password-protected files with strong AES-128, AES-192, and AES-256 encryption.

Starting with version 2.0, zip.hwp also supports some new features of Hollywood 10 like the capability to treat zip archives like file systems which means that you can also use Hollywood functions like DeleteFile() or Rename() directly on files or directories inside zip archives.

1.2 Terms and conditions

WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

1.3 Requirements

- minimum: Hollywood 6.0 or better
- recommended: Hollywood 7.0 or better is recommended because internally zip.hwp uses UTF-8 for all strings and Hollywood versions before 7.0 aren't Unicode-aware
- optimum: Hollywood 10.0 or better because several features of zip.hwp are only available on Hollywood 10 or better

1.4 Installation

Installing zip.hwp is straightforward and simple: Just copy the file `zip.hwp` for the platform of your choice to Hollywood's plugins directory. On all systems except on AmigaOS and compatibles, plugins must be stored in a directory named `Plugins` that is in the same directory as the main Hollywood program. On AmigaOS and compatible systems, plugins must be installed to `LIBS:Hollywood` instead. On macOS, the `Plugins` directory must be inside the `Resources` directory of the application bundle, i.e. inside the `HollywoodInterpreter.app/Contents/Resources` directory. Note that `HollywoodInterpreter.app` is stored inside the `Hollywood.app` application bundle itself, namely in `Hollywood.app/Contents/Resources`.

On Windows you should also copy the file `zip.chm` to the `Docs` directory of your Hollywood installation. Then you will be able to get online help by pressing F1 when the cursor is over a zip.hwp function in the Hollywood IDE.

On Linux and macOS copy the `zip` directory that is inside the `Docs` directory of zip.hwp's distribution archive to the `Docs` directory of your Hollywood installation. Note that on macOS the `Docs` directory is within the `Hollywood.app` application bundle, i.e. in `Hollywood.app/Contents/Resources/Docs`.

2 About zip.hwp

2.1 Credits

zip.hwp was written by Andreas Falkenhahn. Work on this project was started in early 2014 as a proof-of-concept demonstration of Hollywood 6.0's powerful new file and directory adapter plugin interfaces which allow plugins to hook into Hollywood's file and directory handlers. zip.hwp makes use of this feature by making Hollywood think that zip archives are just directories so that it is possible to iterate through them using Hollywood's normal directory functions or open files within zip archives without extracting them first.

If you want to contact me, you can either send an e-mail to `andreas@airsoftsoftwair.de` or use the contact form on `http://www.hollywood-mal.com`.

2.2 Frequently asked questions

This section covers some frequently asked questions. Please read them first before asking on the forum because your problem might have been covered here.

Q: Is there a Hollywood forum where I can get in touch with other users?

A: Yes, please check out the "Community" section of the official Hollywood Portal online at `http://www.hollywood-mal.com`.

Q: Where can I ask for help?

A: There's an active forum at `http://forums.hollywood-mal.com`. You're welcome to join it and ask your question there.

Q: I have found a bug.

A: Please post about it in the "Bugs" section of the forum.

2.3 Known issues

Here is a list of things that zip.hwp doesn't support yet or that may be confusing in some way:

- tbd

2.4 Future

Here are some things that are on my to do list:

- tbd

Don't hesitate to contact me if zip.hwp lacks a certain feature that is important for your project.

2.5 History

Please see the file `history.txt` for a complete change log of zip.hwp.

3 Usage

3.1 Using the plugin

There are several ways of using this plugin: There is a low-level library interface which provides dedicated functions for dealing with zip archives, e.g. zip.OpenArchive() to open an archive and zip.ExtractFile() to extract a file. The low-level interface is a bit more difficult to use than the high-level interface but offers the greatest level of control over zip archives and also the best performance.

Additionally, zip.hwp has a high-level interface which allows your script to deal with zip archives without having to learn any kind of new APIs. Instead, you can just use normal Hollywood functions to access zip archives. There are two ways of using the high-level interface: You can either activate the plugin globally by setting the `InstallAdapter` tag to `True` when `@REQUIRE-ing` it. To do this, simply put the following preprocessor command at the top of your script:

```
@REQUIRE "zip", {InstallAdapter = True}
```

If you activate the high-level interface via `@REQUIRE`, it will become globally available and all ensuing commands that deal with files will support the opening of files from zip archive sources. For example, you could do something like this then:

```
LoadBrush(1, "test.zip/testpicture.jpg")
```

If you only need to open very few files from zip archive sources, you can also choose to not activate the high-level interface globally by omitting the `InstallAdapter` tag on `@REQUIRE` and simply use the `Adapter` tag offered by most Hollywood commands to tell the respective Hollywood command to open the file using the zip.hwp plugin. Here is an example:

```
LoadBrush(1, "test.zip/testpicture.jpg", {Adapter = "zip"})
```

By using the `Adapter` tag, `LoadBrush()` is told to open the specified file using the specified adapter, which is `"zip"` in our case. Thus, the `Adapter` tag allows you to use this plugin even without having installed a global file adapter for it first.

The same is true for Hollywood functions dealing with directories. Once the high-level interface has been activated by setting `InstallAdapter` to `True`, it is possible to do things like the following:

```
OpenDirectory(1, "test.zip")
```

You could then iterate over all files and directories in `test.zip`. If you haven't activated the global adapter for zip.hwp, then just use the local `Adapter` like above, e.g.

```
OpenDirectory(1, "test.zip", {Adapter = "zip"})
```

See the next chapter for more details on treating zip archives as directories.

3.2 Zip archives as directories

When setting the `InstallAdapter` tag to `True`, the zip plugin hooks into Hollywood's file and directory handlers to make Hollywood believe that zip archives are normal directories. This allows you to iterate over all files and directories inside a zip archive using normal functions from Hollywood's DOS library.

For example, to iterate over all files and directory inside a file named `test.zip` you could use the following code:

```
OpenDirectory(1, "test.zip")
Local e = NextDirectoryEntry(1)
While e <> Nil
    DebugPrint(e.name)
    e = NextDirectoryEntry(1)
Wend
CloseDirectory(1)
```

If you don't want to start from the root directory inside `test.zip`, you can also conveniently start from a subdirectory by just pretend that `test.zip` is a directory, e.g. to access a subdirectory named `files` inside `test.zip` just do the following:

```
OpenDirectory(1, "test.zip/files")
```

Finally, it is also possible to recursively iterate through all files and directories inside a zip archive. Here is a function which does that:

```
Function p_DumpZip(d$, idt)
    Local id = OpenDirectory(Nil, d$)
    Local e = NextDirectoryEntry(id)
    While e <> Nil
        If e.Type = #DOSTYPE_DIRECTORY
            DebugPrint(RepeatStr(" ", idt) .. "+", e.name)
            p_DumpZip(FullPath(d$, e.name), idt + 4)
        Else
            DebugPrint(RepeatStr(" ", idt) .. "",e.name,e.size,e.time)
        EndIf
        e = NextDirectoryEntry(id)
    Wend
    CloseDirectory(id)
EndFunction
```

To dump the contents of a zip archive, just call this function like that:

```
p_DumpZip("test.zip", 0)
```

It will then print a nice tree of the zip archive's contents.

3.3 Extracting files

Since zip.hwp hooks into Hollywood's file handler when using the high-level interface, extracting files is just a matter of using Hollywood's `CopyFile()` function on the file you wish to extract. For example, to extract a file named `testpicture.jpg` from `test.zip`, just use the following line:

```
CopyFile("test.zip/testpicture.jpg", "outputdir")
```

If the zip archive is password-protected, you can pass the password to zip.hwp using the new user tags mechanism introduced in Hollywood 10. For example, if `test.zip` uses the password "123456", you could pass that password to zip.hwp like this:

```
CopyFile("test.zip/pic.jpg", "out", {UserTags = {Password = "123456")})
```

Since `CopyFile()` can also copy whole directories including all subdirectories and because zip.hwp hooks into Hollywood's directory handler as well, it is even possible to extract a whole archive using `CopyFile()`, like this:

```
CopyFile("test.zip", "outputdir")
```

You can also open files directly from zip archives because zip.hwp hooks into Hollywood's file handler. For example, you could do this to print all lines of `test.txt` stored in `test.zip`:

```
OpenFile(1, "test.zip/test.txt")
While Not Eof(1) Do DebugPrint(ReadLine(1))
CloseFile(1)
```

All Hollywood functions that deal with files support opening files from zip archives if zip.hwp's file adapter has been enabled. So you could also load images and other data files directly from a zip archive like this:

```
LoadBrush(1, "test.zip/test.jpg")
```

3.4 Modifying archives

You can also modify zip archives by simply using functions from Hollywood's DOS library. For example, to delete the file `test.jpg` from the zip archive `test.zip`, you can just do the following:

```
DeleteFile("test.zip/test.jpg", {Adapter = "zip"})
```

Note that it's mandatory to pass the `Adapter` tag to `DeleteFile()` because zip.hwp doesn't install a filesystem adapter even when setting the `InstallAdapter` tag to `True` (see above). Zip.hwp's filesystem adapter is only accessible by directly passing it to a Hollywood function in the `Adapter` tag.

It's also possible to rename files inside zip archives using Hollywood's `Rename()` function. This can be done like that:

```
Rename("test.zip/oldname.txt", "newname.txt", {Adapter = "zip"})
```

Like above, you have to pass the `Adapter` tag to `Rename()` for this to work.

You can create directories inside zip archives like so:

```
MakeDirectory("test.zip/a_new_dir", {Adapter = "zip"})
```

Don't forget to pass "zip" in the `Adapter` tag here as well.

New files inside zip archives can be created like this:

```
OpenFile(1, "test.zip/new_file", #MODE_WRITE)
WriteLine("Hello World!")
CloseFile(1)
```

Or even shorter:

```
StringToFile("Hello World!", "test.zip/new_file")
```

You can use user tags to specify a password and an encryption level as well:

```
StringToFile("Hello World!", "test.zip/new_file", {UserTags =
      {Password = "123456", Encryption = #ZIP_EM_AES_128}})
```

Note that when writing files to zip archives, existing files in the zip archive won't be deleted but the new files will be appended to the zip archive. Do note, however, that if the file to be written to a zip archive exists, it will be automatically overwritten so be careful.

It's even possible to copy files into zip archives using `CopyFile()`. For example, you could also do something like this:

```
CopyFile("testfile", "test.zip", {Adapter = "zip"})
```

The code above will store the file `testfile` in the zip archive `test.zip`.

Finally, you can change the attributes of files inside zip archives using Hollywood's `SetFileAttributes()` function or move files in and out of zip archives using Hollywood's `MoveFile()` function. Just pass the name of the file you want to modify and `"zip"` in the `Adapter` tag and it will work. It's really convenient and powerful!

3.5 Zip archive basics

When using zip.hwp's low-level interface, you first have to learn some basics about the structure of zip archives.

Zip archives are just a collection of files that are stored at indices ranging from 0 to the number of entries in the zip archive minus 1. It is not necessary to store directories as individual entries. Instead, they can also be stored as part of a filename, e.g. if a file is stored as `a/b/c/test.txt` in the zip archive, then the directories a, b, and c are implicitly declared as existing even though they don't have their individual entries in the zip archive but just exist as part of a file.

Of course, directories can also be stored as individual entries instead of as part of a filename. In that case, they are simply stored as files with a size of 0 bytes with the filename ending in a slash signalling that the entry is a directory. Since there is no distinct directory entry type in zip archives, all functions in this plugin dealing with files can also operate on directories within the zip archive. So don't be confused that a function like zip.RenameFile() can also be used to rename directories and zip.DeleteFile() can also delete directories. Inside a zip archive, directories and files are really pretty much the same except that for directories their filename ends in a slash to signal that it is not a file.

3.6 Creating zip archives

You can create new zip archives by using the zip.OpenArchive(), zip.AddFile(), and zip.CloseArchive() functions. The following code shows how to create a new zip archive named `test.zip` that contains the file `testpicture.jpg`:

```
zip.OpenArchive(1, "test.zip", #MODE_WRITE)
zip.AddFile(1, "testpicture.jpg")
zip.CloseArchive(1)
```

Note that zip.AddFile() does not immediately compress the file and write it to the archive. Instead, files are first collected and they are not compressed and written to the archive before you call zip.CloseArchive(). This is why closing an archive can take quite some time. There is also the possibility to pass a callback function which is invoked by zip.CloseArchive() from time to time so that you can update a status bar or something.

3.7 Linking files

Keep in mind that all files declared in the preprocessor commands are linked automatically into your applet or executable when Hollywood is in compile mode. Thus, if you do some-

thing like the following, not only `testpicture.jpg` but the whole zip archive `test.zip` will be linked to your applet or executable:

```
@BRUSH 1, "test.zip/testpicture.jpg"
```

If you don't want that, you can set the optional `Link` to `False`. If `Link` is set to `False`, Hollywood won't link the specified file to your applet or executable. This means, however, that you have to distribute `test.zip` with your applet or executable so that the data can be loaded from it. Here's how to disable linking:

```
@BRUSH 1, "test.zip/testpicture.jpg", {Link = False}
```

When done like this, Hollywood will never link the file into your applet or executable. Instead, it will always be loaded from the specified file.

4 Function reference

4.1 zip.AddDirectory

NAME

zip.AddDirectory – add directory to zip archive

SYNOPSIS

```
idx = zip.AddDirectory(id, d$[, t])
```

FUNCTION

This functions creates a new directory named `d$` in the zip archive and returns its index. The directory will be empty and you can add files to it using the zip.AddFile() function.

The following tags are currently recognized by the optional table argument:

`Encoding:`

This tag can be used to set the charset encoding that should be used when storing the directory name. This can be one of the following special constants:

`#ZIP_FL_ENC_UTF_8:`

Use UTF-8 encoding. This is the default.

`#ZIP_FL_ENC_CP437:`

Use code page 437 encoding. Since this was the standard encoding on MS-DOS it was also the standard encoding of the original zip format. So if you need maximum compatibility, you can use this encoding but remember that it can only store Western characters.

(V1.2)

INPUTS

`id`	identifier of the zip archive to use
`d$`	name of the directory to create in the zip archive
`table`	optional: table containing further options (see above) (V1.2)

RESULTS

`idx`	index of newly added directory in zip archive

4.2 zip.AddFile

NAME

zip.AddFile – add file to zip archive

SYNOPSIS

```
idx = zip.AddFile(id, f$[, table])
```

FUNCTION

This function adds the file specified by `f$` to the zip archive specified by `id` and returns the index of the newly added file. The optional table argument allows you to specify further options.

The following tags are currently recognized by the optional table argument:

`NewName:` This tag allows you to store this file with a new name in the zip archive. If you want to store the file in a subdirectory in the zip archive, you also have to use this tag and include the name of the subdirectory(s) in `NewName`. If `NewName` is omitted, the file will always be stored in the root directory of the zip archive.

`Encryption:`
This tag allows you to set the desired encryption method for the file. It can be set to one of the following special constants:

`#ZIP_EM_NONE:`
No encryption. This is the default.

`#ZIP_EM_AES_128:`
Winzip AES-128 encryption.

`#ZIP_EM_AES_192:`
Winzip AES-192 encryption.

`#ZIP_EM_AES_256:`
Winzip AES-256 encryption.

If you specify the `Encryption` tag, it is also necessary to provide a password that is needed to decrypt the file. You can specify this password in the `Password` tag (see below). If you don't use the `Password` tag, the default password set using zip.SetDefaultPassword() is used.

`Password:`
If the `Encryption` tag has been set to a value different from `#ZIP_EM_NONE` (see above), this tag can be set to a password that should be used to protect the file. If you omit this tag, the default password set using zip.SetDefaultPassword() is used.

`Compression:`
This tag can be used to set the desired compression method for the file. The following compression methods are currently supported:

`#ZIP_CM_DEFAULT:`
This is the default setting. Currently the same as `#ZIP_CM_DEFLATE`.

`#ZIP_CM_STORE:`
Store the file uncompressed.

`#ZIP_CM_BZIP2:`
Compress the file using the bzip2 algorithm.

`#ZIP_CM_DEFLATE:`
Deflate the file with the zlib algorithm and default options.

Note that only `#ZIP_CM_DEFLATE` and `#ZIP_CM_STORE` can be assumed to be universally supported.

When specifying this tag, you can also pass the `CompressionFlags` tag to set the compression level (see below).

`CompressionFlags:`
This tag can be used to define the compression level. It ranges from 1 for the fastest compression and 9 for the highest. You can also pass 0 to use the compressor's default setting. Defaults to 0.

`Comment:` This tag can be used to add the file to the zip archive with a comment attached to it.

`Time:` This tag can be used to change the file's datestamp. By default, the datestamp will be taken from the file specified in `f$`. If you'd like to assign a different datestamp to the file, then you need to pass a string in the standard Hollywood date format of `dd-mmm-yyyy hh:mm:ss` to this tag.

`Encoding:`
This tag can be used to set the charset encoding that should be used when storing the filename. This can be one of the following special constants:

`#ZIP_FL_ENC_UTF_8:`
Use UTF-8 encoding. This is the default.

`#ZIP_FL_ENC_CP437:`
Use code page 437 encoding. Since this was the standard encoding on MS-DOS it was also the standard encoding of the original zip format. So if you need maximum compatibility, you can use this encoding but remember that it can only store Western characters.

(V1.2)

Note that this function doesn't immediately begin compressing the file and writing it to the zip archive. Instead, the file is just added into an internal list and compressing and writing will be done once you call zip.CloseArchive(). This means that you have to make sure that the file you specified in `f$` is still available when you call zip.CloseArchive(), i.e. in case you pass the name of a temporary file to `f$` you must not delete this temporary file before you call zip.CloseArchive().

INPUTS

`id` identifier of the zip archive to use

`f$` path to a file to add to the zip archive

`table` optional: table containing further options (see above)

RESULTS

`idx` index of newly added file in zip archive

4.3 zip.CloseArchive

NAME

zip.CloseArchive – close zip archive

SYNOPSIS

```
zip.CloseArchive(id[, discard, callback, userdata])
```

FUNCTION

This function closes the specified zip archive. Note that if the zip archive has been opened for writing, `zip.CloseArchive()` marks the point when compressing and writing the data will actually happen. That's why it can take some time for this function to return.

If you want to discard all changes that have been made to the zip archive, you have to pass `True` in the `discard` parameter. In that case, the original zip archive isn't modified and all changes are discarded. This is also what will happen to all zip archives which you open using zip.OpenArchive() but forget to close using `zip.CloseArchive()`. Changes will only ever be written to the zip archive if you explicitly call `zip.CloseArchive()` with `discard` being `False`.

If you'd like to monitor the progress of compressing data and writing it to the zip archive, you can pass a callback function in the second parameter. Optionally, it is also possible to specify user data to pass to the callback function in their third argument. The `userdata` parameter can take values of any type: Numbers, strings, tables, and even functions can be passed as user data.

The status callback function receives a single table element that contains the following fields:

`Action:` Initialized to `"CloseArchive"`.

`ID:` Contains the identifier of the zip archive that is currently being worked on.

`Progress:`
Contains a value between 0 and 100 indicating how much work has already been done.

`UserData:`
Contains the value you passed in the `userdata` argument.

Obviously, if `discard` is set to `True`, the callback function will never be called.

INPUTS

`id` identifier of the zip archive to be closed

`discard` optional: `True` to discard all changes, `False` to write all changes to the zip archive (defaults to `False`)

`callback` optional: function to call from time to time

`userdata` optional: user specific data to pass to callback function

4.4 zip.DeleteFile

NAME

zip.DeleteFile – delete file from zip archive

SYNOPSIS

`zip.DeleteFile(id, idx)`

FUNCTION

This function deletes the file at index `idx` in the zip archive specified by `id`.

Note that the change to the zip archive isn't done immediately but is postponed until you call zip.CloseArchive().

This function can also operate on directories. See Section 3.5 [Zip archive basics], page 8, for details.

INPUTS

`id`	identifier of the zip archive to use
`idx`	file to delete

4.5 zip.ExtractFile

NAME

zip.ExtractFile – extract file from zip archive

SYNOPSIS

`zip.ExtractFile(id, idx, dst$[, table])`

FUNCTION

This function can be used to extract the file at the index `idx` inside the zip archive specified by `id` to the external file specified by `dst$`. If `dst$` already exists, it will be overwritten. An optional table argument allows you to specify further options for the operation.

The following tags are currently recognized in the optional table argument:

`Password:`

If the file you wish to extract is protected by a password, you have to specify this password here. If you don't specify this tag, the default password set using zip.SetDefaultPassword() is used.

`Callback:`

This tag allows you to pass a function that should be called from time to time. This can be useful if you'd like to show a status bar or something while the zip file is being extracted. The function will receive a table as its sole argument. The table will have the following fields initialized:

`Action:`	Initialized to `"ExtractFile"`.
`ID:`	Contains the identifier of the zip archive.
`Progress:`	Contains a value between 0 and 100 indicating how much work has already been done.

`UserData:`
: Contains the value you passed in the `UserData` argument (see below).

You can also pass user data that should be forwarded to your callback using the tag below.

`UserData:`
: This tag can be set to arbitrary data that should be passed to the callback you passed in the `Callback` tag. If you specify this tag without the `Callback` tag, it is simply ignored.

INPUTS

`id`	identifier of the zip archive to use
`idx`	index of file to extract
`dst$`	desired destination file
`table`	optional: table containing further parameters

4.6 zip.GetFileAtIndex

NAME

zip.GetFileAtIndex – get name of file by index

SYNOPSIS

```
name$ = zip.GetFileAtIndex(id, idx)
```

FUNCTION

This function returns the name of the file at index `idx` in the zip archive specified by `id`. If the name returned by this function ends with a slash, it is a directory, otherwise it is a file. See Section 3.5 [Zip archive basics], page 8, for details.

To find out the number of files in a zip archive, you can query `#ZIPATTRNUMENTRIES` with Hollywood's `GetAttribute()` function. See Section 4.9 [zip.GetObjectType], page 18, for details.

INPUTS

`id`	identifier of the zip archive to use
`idx`	index to query (in the range of 0 to number of entries minus 1)

RESULTS

`name$`	name of entry at index

4.7 zip.GetFileAttributes

NAME

zip.GetFileAttributes – get file attributes

SYNOPSIS

```
t = zip.GetFileAttributes(id, idx)
```

FUNCTION

This function returns attributes of the file at the index specified by `idx` inside the zip archive specified by `id`. `zip.GetFileAttributes()` returns a table with the following information about the file:

`Size:` The size of the file in bytes or 0 for directories.

`CompressedSize:`
The compressed size of the file in bytes or 0 for directories.

`CRC32:` The CRC32 checksum of the file or 0 for directories.

`Compression:`
The compression method used for the file. This will be one of the following special constants:

`#ZIP_CM_DEFAULT:`
Default compression. Currently the same as `#ZIP_CM_DEFLATE`.

`#ZIP_CM_STORE:`
Store the file uncompressed.

`#ZIP_CM_BZIP2:`
Compress the file using the bzip2 algorithm.

`#ZIP_CM_DEFLATE:`
Deflate the file with the zlib algorithm and default options.

`Encryption:`
The encryption method used for the file. This will be one of the following special constants:

`#ZIP_EM_NONE:`
No encryption.

`#ZIP_EM_AES_128:`
Winzip AES-128 encryption.

`#ZIP_EM_AES_192:`
Winzip AES-192 encryption.

`#ZIP_EM_AES_256:`
Winzip AES-256 encryption.

`Time:` The datestamp for the file. This will be in the standard Hollywood date format of `dd-mmm-yyyy hh:mm:ss`.

INPUTS

`id` identifier of the zip archive to use

`idx` index of the file to query

RESULTS

`t` table containing file attributes

4.8 zip.GetFileComment

NAME

zip.GetFileComment – get file comment

SYNOPSIS

```
c$ = zip.GetFileComment(id[, idx])
```

FUNCTION

This function can be used to get the comment of a file inside the zip archive specified by `id` or of the whole zip archive. If the optional parameter `idx` is specified, `zip.GetFileComment()` retrieves the comment of the file at that index. If the `idx` parameter is omitted or set to -1, the comment of the zip archive itself is returned.

This function can also operate on directories. See Section 3.5 [Zip archive basics], page 8, for details.

INPUTS

`id` identifier of the zip archive to use

`idx` optional: index of file whose comment should be obtained; if this is omitted or set to -1, the comment of the whole archive is returned (defaults to -1)

RESULTS

`c$` file comment or empty string if there is no comment

4.9 zip.GetObjectType

NAME

zip.GetObjectType – get zip archive object type

SYNOPSIS

```
type = zip.GetObjectType()
```

FUNCTION

This function returns the object type used by zip archives opened using the zip.OpenArchive() function. You can then use this object type with functions from Hollywood's object library such as `GetAttribute()`, `SetObjectData()`, `GetObjectData()`, etc.

In particular, Hollywood's `GetAttribute()` function may be used to query certain properties of zip archives opened using zip.OpenArchive(). The following attributes are currently supported by `GetAttribute()` for zip archives:

`#ZIPATTRNUMENTRIES:`
: Returns the number of entries in the zip archive.

INPUTS

none

RESULTS

`type` internal zip archive type for use with Hollywood's object library

EXAMPLE

```
zip.OpenArchive(1, "test.zip")
ZIP_ARCHIVE = zip.GetObjectType()
numentries = GetAttribute(ZIP_ARCHIVE, 1, #ZIPATTRNUMENTRIES)
```

The code above opens `test.zip` and queries the number of entries in the archive via `GetAttribute()`.

4.10 zip.LocateFile

NAME

zip.LocateFile – find file in zip archive

SYNOPSIS

```
idx = zip.LocateFile(id, name$[, table])
```

FUNCTION

This function searches for the file specified by `name$` inside the zip archive specified by `id` and returns its index if it is found, otherwise -1 is returned.

The optional table argument can be used to specify further options. The following table tags are currently recognized:

`NoCase:` If this tag is set to `True`, `zip.LocateFile()` won't distinguish between upper and lower case characters. This makes the search slower. Defaults to `False`.

`NoDir:` If this tag is set to `True`, `zip.LocateFile()` will just match the file name so it will also trigger if the file is in a subdirectory in the archive. Defaults to `False`.

This function can also operate on directories. See Section 3.5 [Zip archive basics], page 8, for details.

INPUTS

`id`	identifier of the zip archive to use
`name$`	name of the file to locate
`table`	optional: table argument containing further options (see above)

RESULTS

`idx`	index of file inside zip archive or -1 if it couldn't be found

4.11 zip.OpenArchive

NAME

zip.OpenArchive – open a zip archive for reading or writing

SYNOPSIS

```
[id] = zip.OpenArchive(id, filename$[, mode])
```

FUNCTION

This function attempts to open the zip archive specified by `filename$` and assigns `id` to it. If you pass `Nil` in `id`, `zip.OpenArchive()` will automatically choose a vacant identifier and return it. If the file does not exist, this function will fail unless you use the mode argument to open a zip archive for writing. In that case, `zip.OpenArchive()` will create the file for you.

The following modes are currently supported:

`#MODE_READ:`
: Open the zip archive for reading. This is the default mode.

`#MODE_READWRITE:`
: Open the zip archive for reading and writing. If the specified zip archive doesn't exist, it is automatically created.

`#MODE_WRITE:`
: Open the zip archive for writing. If the specified zip archive already exists, it will be overwritten.

Although zip.hwp will automatically close all open zip archives when it quits, it is strongly advised that you close an open zip archive when you are done with it using the zip.CloseArchive() function because otherwise you are wasting resources and in case you are writing or modifying a zip archive, zip.CloseArchive() is where the actual work is done.

Note that `zip.OpenArchive()` will create a standard Hollywood object which can also be used with functions from Hollywood's object library such as `GetAttribute()`, `SetObjectData()`, `GetObjectData()`, etc. See Section 4.9 [zip.GetObjectType], page 18, for details.

INPUTS

`id` identifier of the file or `Nil` for auto id selection

`filename$`
: name of the file to open

`mode` mode to open the file; can be `#MODE_READ`, `#MODE_WRITE` or `#MODE_READWRITE` (defaults to `#MODE_READ`)

RESULTS

`id` optional: identifier of the file; will only be returned when you pass `Nil` as argument 1 (see above)

4.12 zip.RenameFile

NAME

zip.RenameFile – rename file in zip archive

SYNOPSIS

```
zip.RenameFile(id, idx, newname$[, t])
```

FUNCTION

This function renames the file at index `idx` in the zip archive specified by `id` to the name passed in the `newname$` parameter.

If you need to rename a directory entry, `newname$` must end on a trailing slash character. See Section 3.5 [Zip archive basics], page 8, for details.

Note that the change to the zip archive isn't done immediately but is postponed until you call zip.CloseArchive().

The following tags are currently recognized by the optional table argument:

`Encoding:`
: This tag can be used to set the charset encoding that for the filename. This can be one of the following special constants:

  `#ZIP_FL_ENC_UTF_8:`
  : Use UTF-8 encoding. This is the default.

  `#ZIP_FL_ENC_CP437:`
  : Use code page 437 encoding. Since this was the standard encoding on MS-DOS it was also the standard encoding of the original zip format. So if you need maximum compatibility, you can use this encoding but remember that it can only store Western characters.

  (V1.2)

INPUTS

`id`	identifier of the zip archive to use
`idx`	file to rename
`newname$`	new name for file
`table`	optional: table containing further options (see above) (V1.2)

4.13 zip.SetDefaultPassword

NAME

zip.SetDefaultPassword – set default password

SYNOPSIS

```
zip.SetDefaultPassword(id, pwd$)
```

FUNCTION

This function can be used to set a default password that is used to encrypt and decrypt files if no other password is provided. You need to pass the identifier of a zip archive in `id` and a password in `pwd$`. If you pass an empty string in `pwd$`, the default password is unset.

Functions that use the default password if no other password is explicitly specified are zip.AddFile(), zip.ExtractFile(), and zip.SetFileEncryption().

INPUTS

`id`	identifier of the zip archive to use
`pwd$`	new default password or empty string to unset the default password

4.14 zip.SetFileComment

NAME

zip.SetFileComment – set file comment

SYNOPSIS

```
zip.SetFileComment(id[, c$, idx])
```

FUNCTION

This function can be used to set the comment for a file inside the zip archive specified by `id` or for the whole zip archive. If the optional parameter `idx` is specified, the comment specified by `c$` is set for the file at that index. If the optional parameter `idx` is omitted or set to -1, the comment specified by `c$` is set for the whole zip archive.

You can also remove the comment of a file or the whole archive by omitting the `c$` argument or setting it to an empty string.

Note that the change to the zip archive isn't done immediately but is postponed until you call zip.CloseArchive().

This function can also operate on directories. See Section 3.5 [Zip archive basics], page 8, for details.

INPUTS

`id`	identifier of the zip archive to use
`c$`	optional: comment to set or empty string to remove a comment
`idx`	optional: index of file whose comment should be set; if this is omitted or set to -1, the comment is set for the whole archive (defaults to -1)

4.15 zip.SetFileCompression

NAME

zip.SetFileCompression – set file compression

SYNOPSIS

```
zip.SetFileCompression(id, idx, method[, flags])
```

FUNCTION

This function sets the compression method for the file at index `idx` in the zip archive specified by `id` to the compression method specified in `method`. The optional `flags` argument can be used to define the compression level, 1 being fastest compression and 9 highest. Allowed values are 1-9 or 0 to use the compressor's default setting.

The `method` parameter must be one of the following constants:

`#ZIP_CM_DEFAULT`:
: This is the default setting. Currently the same as `#ZIP_CM_DEFLATE`.

`#ZIP_CM_STORE`:
: Store the file uncompressed.

`#ZIP_CM_BZIP2`:
: Compress the file using the bzip2 algorithm.

`#ZIP_CM_DEFLATE`:
: Deflate the file with the zlib algorithm and default options.

Note that only `#ZIP_CM_DEFLATE` and `#ZIP_CM_STORE` can be assumed to be universally supported.

Also note that the change to the zip archive isn't done immediately but is postponed until you call zip.CloseArchive().

INPUTS

`id`	identifier of the zip archive to use
`idx`	index of file whose compression should be set
`method`	desired compression method (see above)
`flags`	optional: desired compression level ranging from 1 (fastest) to 9 (highest) or 0 for the compressor's default setting (defaults to 0)

4.16 zip.SetFileEncryption

NAME

zip.SetFileEncryption – set file encryption

SYNOPSIS

```
zip.SetFileEncryption(id, idx, method[, pwd$])
```

FUNCTION

This function sets the encryption method for the file at index `idx` in the zip archive specified by `id`. The desired encryption method has to be passed in the `method` parameter. Optionally, you can specify a password in the `pwd$` argument. If the `pwd$` argument is omitted or set to an empty string, the default password set using zip.SetDefaultPassword() is used.

The `method` parameter must be one of the following constants:

`#ZIP_EM_NONE`:
: No encryption.

`#ZIP_EM_AES_128`:
: Winzip AES-128 encryption.

`#ZIP_EM_AES_192`:
: Winzip AES-192 encryption.

`#ZIP_EM_AES_256`:
: Winzip AES-256 encryption.

Note that the change to the zip archive isn't done immediately but is postponed until you call zip.CloseArchive().

INPUTS

`id`	identifier of the zip archive to use
`idx`	index of file whose encryption should be set
`method`	desired encryption method (see above)
`pwd$`	optional: desired password for file or empty string to use the default password (defaults to the empty string)

4.17 zip.SetFileTime

NAME

zip.SetFileTime – set file datestamp

SYNOPSIS

```
zip.SetFileTime(id, idx, time$)
```

FUNCTION

This function sets the datestamp for the file at index `idx` in the zip archive specified by `id`. The datestamp must be passed in the standard Hollywood date format of `dd-mmm-yyyy hh:mm:ss`.

Note that the change to the zip archive isn't done immediately but is postponed until you call zip.CloseArchive().

INPUTS

`id`	identifier of the zip archive to use
`idx`	index of file whose datestamp should be changed
`time$`	desired datestamp

Appendix A Licenses

A.1 LibZip license

The authors can be contacted at <libzip@nih.at>

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of the authors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHORS AS IS AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

A.2 AES encryption support

The free distribution and use of this software in both source and binary form is allowed (with or without changes) provided that:

1. distributions of this source code include the above copyright notice, this list of conditions and the following disclaimer;
2. distributions in binary form include the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other associated materials;
3. the copyright holder's name is not used to endorse products built using this software without specific written permission.

ALTERNATIVELY, provided that this notice is retained in full, this product may be distributed under the terms of the GNU General Public License (GPL), in which case the provisions of the GPL apply INSTEAD OF those given above.

DISCLAIMER: This software is provided 'as is' with no explicit or implied warranties in respect of its properties, including, but not limited to, correctness and/or fitness for purpose.

Issue Date: 18th November 2008

Index

zip.AddDirectory 11
zip.AddFile 11
zip.CloseArchive 13
zip.DeleteFile 14
zip.ExtractFile 15
zip.GetFileAtIndex 16
zip.GetFileAttributes 16
zip.GetFileComment 17
zip.GetObjectType 18
zip.LocateFile 19
zip.OpenArchive 19
zip.RenameFile 20
zip.SetDefaultPassword 21
zip.SetFileComment 22
zip.SetFileCompression 22
zip.SetFileEncryption 23
zip.SetFileTime 24