Pawn

embedded scripting language

File I/O Support Library

Contents

January 2015 CompuPhase

"CompuPhase" and "Pawn" are trademarks of ITB CompuPhase.
"Java" is a trademark of Sun Microsystems, Inc.
"Linux" is a registered trademark of Linus Torvalds.
"Microsoft" and "Microsoft Windows" are registered trademarks of Microsoft Corporation.
"Unicode" is a trademark of Unicode, Inc.

Copyright © 2004–2015, ITB CompuPhase
Eerste Industriestraat 19–21, 1401VL Bussum The Netherlands
telephone: (+31)-(0)35 6939 261
e-mail: info@compuphase.com
WWW: http://www.compuphase.com

The information in this manual and the associated software are provided "as is". There are no guarantees, explicit or implied, that the software and the manual are accurate.

Typeset with TEX in the "DejaVu" typeface family.

Introduction

The "PAWN" programming language depends on a host application to provide an interface to the operating system and/or to the functionality of the application. This interface takes the form of "native functions", a means by which a PAWN script calls into the application. The PAWN "core" toolkit mandates or defines *no* native functions at all (the tutorial section in the manual uses only a *minimal* set of native functions in its examples). In essence, PAWN is a bare language to which an application-specific library must be added.

That notwithstanding, the availability of general purpose native-function libraries is desirable. In this view, I developed the file input/output support library for general purpose reading and writing to text and binary files.

This application note assumes that the reader understands the PAWN language. For more information on PAWN, please read the manual "The PAWN booklet — The Language" which is available from the company homepage.

Platform differences

Operating systems differ in their conventions for file/path names and the encoding of text files. The file I/O library addresses these platform differences to some extent, in order to allow portable PAWN scripts.

UNIX and UNIX-like operating systems use a forward slash to separate names of directories and files, whereas Microsoft DOS and Windows use a backslash and the Apple Macintosh uses a colon. The file I/O library accepts paths in the "native OS" format as well as in the UNIX format. Forward slashes in path names are automatically translated to the proper directory separator for the operating system.

Note that under Microsoft DOS, file and directory names are still limited to eight characters plus an extension of three characters. In addition, there is no portable method for specifying a drive (e.g. "`A:\myfile.txt`"). Drive specifications are typically ignored, however —see the section "Security".

UNIX uses a single "line feed" character to end a text line (ASCII 10), Apple Macintosh uses a "carriage return" character (ASCII

13) and Microsoft DOS/Windows use the pair of carriage return and line feed characters. Many high-level protocols of the TCP/IP protocol suite also require both a carriage return and a line feed character to end a line —examples are RFC 854 for Telnet, RFC 821 for SMTP and RFC 2616 for HTTP.

The file I/O support library provides functions for reading and writing lines and blocks from/to a file. The line reading/writing functions are for text files and the block reading/writing functions for binary files. Additional functions allow you to read/write character by character or or byte by byte; these functions are indifferent for text versus binary files.

The line reading functions, `fread` and `fwrite`, check for all three common line ending specifications: CR, LF and CR–LF. If a LF character follows a CR character, it is read and considered part of a CR–LF sequence; when any other character follows CR, the line is assumed to have ended on the CR character. This implies that you cannot embed single CR characters in a DOS/Windows or UNIX file, and neither use LF characters in lines in a Macintosh file. It is uncommon, though, that such characters appear. The pair LF–CR (CR–LF in the inverted order) is *not* supported as a valid line-ending combination.

The line writing function writes the characters as they are stored in the string. If you wish to end lines with a CR–LF pair, you should end the string to write with `\r\n`.

The line reading and writing functions support UTF-8 encoding when the string to read/write is in *unpacked* format. When the source or destination string is a *packed* string, the line functions assume ASCII or another 8-bit encoding —such as one of the ISO/IEC 8859 character sets (ISO/IEC 8859-1 is informally known as “Latin-1”). Please see the manual “The PAWN booklet — The Language” for details on packed and unpacked strings.

The block reading and writing functions, `fblockread` and `fblockwrite`, transfer the specified number of cells as a binary block. The file is assumed to be in Little Endian format (Intel byte order). On a Big Endian microprocessor, the block reading/writing functions translate the data from Big Endian to Little Endian on the flight.

The character reading/writing functions, `fgetchar` and `fputchar`, read and write a single byte respectively. Byte order considerations are irrelevant. These functions apply UTF-8 encoding by default, but they can also read/write raw bytes.

Next to data transfer functions, the library contains file support functions for opening and closing files (`fopen`, `fclose`), checking whether a file exists, (`fexist`), browsing through files (`fexist` and `fmatch`), deleting a file (`fremove`), creating a temporary file which is automatically deleted when you close it (`ftemp`), and modifying the current position in the file (`fseek`).

Filename matching

The filename matching functions `fmatch` and `fexist` support filenames with "wild-card" characters —also known as filename patterns. The concept of these patterns exists in all contemporary operating systems (such as Microsoft Windows and UNIX/Linux), but they differ in minor ways in which characters they use for the wild-cards.

The patterns described here are a simplified kind of "regular expressions" found in compiler technology and some developer's tools. The patterns do not have the power or flexibility of full regular expressions, but they are simpler to use.

Patterns are composed of normal and special characters. Normal characters are letters, digits, and other a set of other characters; actually, everything that is not a *special* character is "normal". The special characters are discussed further below. Each normal character matches one and only one character —the character itself. For example, the normal character "a" in a pattern matches the letter "a" in a name or string. A pattern composed entirely of normal characters is a special case since it matches only one exactly one name/string: all characters must match exactly. The empty string is also a special case, which matches only empty names or strings.

Some operating systems (such as UNIX) support case-sensitive filenames, so that the names "abc", "ABC", and "Abc" all refer to different files. Others (such as DOS and Windows) support case-insensitive filenames, so that the previous names all refer to the same file.

Special pattern characters match *any* character, int single or multiple occurrences, or only a selected set of characters. The special pattern characters are:

? Any
   The *any* pattern ? matches any single character.

`*` Closure
The *closure* pattern * matches zero or more non-specific characters.

`[abc]` Set
The *set* pattern `[abc]` matches a single character in the set (`a`, `b`, `c`). On case-insensitive matches, this will also match any character in the set (`A`, `B`, `C`). If the set contains the `]` character, it must be quoted (see below). If the set contains the hyphen character `-`, it must be the first character in the set, be quoted, or be specified as the range `---`.

`[a-z]` Range set
The *range* pattern `[a-z]` matches a single character in the range `a` through `z`. On case-insensitive matches, this will also match any character in the range `A` through `Z`. The character before the hyphen must sort lexicographically before the character after the hyphen. Sets and ranges can be combined within the same set of brackets; e.g. the pattern `[a-c123]` matches any character in the set (`a`, `b`, `c`, `1`, `2`, `3`).

`[!abc]` Excluded set
The *excluded set* pattern `[!abc]` matches a single character not in the set (`a`, `b`, `c`). Case-insensitive systems also exclude characters in the set (`A`, `B`, `C`). If the set contains the hyphen character, it must immediately follow the `!` character, be quoted, or be specified as the range `---`. In any case, the `!` must immediately follow the `[` character.

`{abc}` Repeated set
The *repeated set* is similar to the normal set, `[abc]`, except that it matches zero or more occurrences of the characters in the set. It is similar to a *closure*, but matching only a subset of all characters. Similar to single character sets, the repeated set also supports ranges, as in `{a-z}`, and exclusions, as in `{!abc}`.

`` `x `` Quoted (literal) character
A *back-quote* character ` removes any special meaning from the next character. To match the quote character itself, it must be quoted itself, as in ``` `` ```. The back-quote followed by two hexadecimal digits gives the character with the byte value of the hexadecimal number. This can be used to insert any character value in the string, including the binary zero. The back-quote character is also called the *grave accent*.

Some patterns, such as *, would match empty names or strings. This is generally undesirable, so empty names are handled as a special case, and they can be matched only by an empty pattern.

PAWN uses the zero character as a string terminator. To match a zero byte, you must use `` `00 `` in the pattern. For example, the pattern `a[`00-`1f]` matches a string that starts with the letter "a" followed by a byte with a value between 0 and 31.

INI files

Many programs need to store settings between sessions. For this reason, the library provides a set of high-level functions for storing the configuration in an "INI" file. An INI file is a plain text file where fields are stored as name/value pairs. The name (called the "key" in the function descriptions) and the value are separated by an equal sign ("=") or a colon (":") —the colon separator is an extension of this library.

INI files are optionally divided into sections. A section starts with a section name between square brackets.

INI files are best known from Microsoft Windows, but several UNIX and Linux programs also use this format (although the file extension is sometimes ".cfg" instead of ".ini"). Playlist files in Shoutcast/Icecast format also use the syntax of INI files.

UTF-8

UTF-8 is a variable length symbol encoding, for storing texts in Unicode or other multi-byte character sets. ISO/IEC 10646-1 standardizes the Universal Character Set (UCS) in two encodings: a 4-byte encoding called UCS-4 and a 2-byte encoding called UCS-2. UCS-2 is the currently same as Unicode and it contains (roughly) the first 64,000 characters of the UCS: the Basic Multilingual Plane (BMP).

UTF-8 is stores the UCS-4 set in 1 to 6 bytes per character and the UCS-2 set in 1 to 3 bytes per character. The UTF-8 encoding is becoming a popular replacement for ASCII, especially for porting TCP/IP protocols to wider character sets. The RFC 2279 describes the UTF-8 encoding; the official standard is ISO/IEC 10646-1, annex R.

UTF-8 is fully compatible with 7-bit ASCII. It is, however, not compatible with the *extended* ASCII character sets, like ISO/IEC 8859. For example, the letter å is represented in ISO-8859-1 (Latin-1) as E5 (hexadecimal, or 229 in decimal), but when this is encoded as UTF-8 it is represented by the two bytes C3–A5 (hexadecimal). The lowest 256 codes of the Unicode set, by the way, are the same as those of ISO-8859-1; that is, the character å is represented in Unicode as U+00E5 —the same numerical value as it has in ISO-8859-1.

The UTF-8 encoding is very regular and contains sufficient redundancy to make the chances of heuristic detection of UTF-8 very high. That is, if a text passes the validation tests of UTF-8 encoding, it is very likely that the text is indeed UTF-8. For instance, two-byte "leader" codes are in the range 192–223 and "follower" codes range 128–191. In ISO-8859-1, nearly all leader codes represent accented capitals while the range for the follower codes is for special symbols (non-letters). The chance that an upper case accented letter is followed by a special symbol is very small in European languages.

An additional advantage is that the UTF-8 encoding scheme is the same irrespective of whether the underlying processor is Little Endian or Big Endian. No Byte Order Mark (BOM) is required at the start of a message or text. That said, some applications write a BOM at the start of an UTF-8 file to mark the file as UTF-8 (as opposed to plain ASCII or Latin-1).

Some languages/libraries/papers implement or propose minor modifications to UTF-8. For example, Java uses a 2-byte code to store ASCII zero whereas only a single byte is required. Although this may seem to be just inefficient storage, the UTF-8 standard is quite explicit in its insistence that values should be encoded in the most compact form. The reason is that inefficient storage harms the heuristics for distinguishing UTF-8 from an 8-bit encoding (for example, Latin-1), and it introduces security weaknesses. The widespread practice storing generating surrogate pairs (the encoding of a 4-byte sequence by two 2-byte sequences) as two UTF-8 characters is also *invalid* UTF-8.

The file I/O library heuristically detects whether a line that it reads is UTF-8 or not. If the line cannot be interpreted as UTF-8, it is, of course, not UTF-8 and it is assumed to be an 8-bit ISO-8859 encoding. If the line adheres to the syntax rules of UTF-8, interpreted strictly, the line is seen as UTF-8. ASCII is always interpreted correctly, because UTF-8 is fully compatible

with 7-bit ASCII. The line reading/writing functions support UTF-8 only when the source/destination string is an *unpacked* string, because only unpacked strings can store the full UCS character set.

The file I/O support library does neither requires a Byte Order Mark (BOM), nor interprets it in any special way. When a BOM is present in the file, it is read like a common Unicode character.

The strict interpretation of the UTF-8 syntax rules may cause it to fail reading UTF-8 files generated by non-conforming applications. Writing overly long sequences (as Java does with the null character) and incorrect encoding of surrogate pairs were already mentioned, but other non-conforming implementations exist as well. The UTF-8 standard advises to insert a special "invalid symbol" character in the stream when reading an invalid code sequence, but this library falls back to interpreting the string as non-UTF-8 instead.

Security

The file I/O support library provides functions to overwrite and remove files. To allow untrusted scripts to use files, the file I/O support library restricts file access to only a specific directory. This directory name is in an environment variable, whose default name is AMXFILE.* If that setting is absent, the file I/O library uses the directory indicated by the TMP, TEMP or TMPDIR environment variables. If these are absent too, every file access or removal attempt fails.

The paths that you use to access a file, e.g. in the native function `fopen`, are prefixed by the directory mentioned by the "AMXFILE" environment variable. Any root directory specifications or drive letters in the file path are ignored.

For example, if the AMXFILE environment variable is set to `/tmp`, the path `local/myfile.txt` maps to `/tmp/local/myfile.txt`. Absolute paths and UNC paths are handled too: the path `//mybox/local/myfile.txt` will still refer to `/tmp/local/myfile.txt` (even if `mybox` refers to a different host than the current machine).

---

* The actual name depends on how the library is implemented, see the chapter "Implementing the library".

The examples above use the forward slash as the directory separator, but the native OS directory separator is handled in the same way.

You can set the AMXFILE environment variable to the root directory of the local drive, giving the file I/O support library access to any file on the system, but this is not advised. When you install the file I/O support library, it is advised that you verify that the security system is in place and working correctly. For example, the following script should write the file "`testfile.txt`" in the directory set in the AMXFILE environment variable or in the "temporaries" directory, but *not* in the root directory.

LISTING: script to test whether the root directory is shielded

```
#include <file>

main()
    {
    new File: file = fopen("/testfile.txt", io_write)
    if (file)
        {
        fwrite file, "hello world\n"
        fclose file
        print "Please verify that the \"testfile.txt\" file is \
               not in the root directory.\n"
        }
    else
        print "Failed to create the file \"testfile.txt\".\n"
    }
```

As explained in the section on UTF-8, the file I/O support library uses a strict interpretation of the UTF-8 encoding format. This is partly done for reasons of guarding against deliberately malformed UTF-8 strings.

Implementing the library

The file I/O support library consists of the files AMXFILE.C and FILE.INC. The C file may be "linked in" to a project that also includes the PAWN abstract machine (AMX.C), or it may be compiled into a DLL (Microsoft Windows) or a shared library (Linux). The .INC file contains the definitions for the PAWN compiler of the native functions in AMXFILE.C. In your PAWN programs, you may either include this file explicitly, using the `#include` preprocessor directive, or add it to the "prefix file" for automatic inclusion into any PAWN program that is compiled.

The "Implementer's Guide" for the PAWN toolkit gives details for implementing the extension module described in this application note into a host application. The initialization function, for registering the native functions to an abstract machine, is `amx_FileInit` and the "clean-up" function is `amx_FileCleanup`. In the current implementation, calling the clean-up function is not required.

If the host application supports dynamically loadable extension modules, you may alternatively compile the C source file as a DLL or shared library. No explicit initialization or clean-up is then required. Again, see the Implementer's Guide for details.

The C source code contains a variable name and conditionally compiled code that can be configured via a compiler option. The preprocessor macro `AMXFILE_VAR` allows you to set the name of environment variable that specifies the restricted directory (see the section "Security"). The default value for this macro is "`AMXFILE`". If you set this macro to an empty string when compiling, the security features of the file I/O support library are removed.

Usage

Depending on the configuration of the PAWN compiler, you may need to explicitly include the FILE.INC definition file. To do so, insert the following line at the top of each script:

```
#include <file>
```

The angle brackets "<...>" make sure that you include the definition file from the system directory, in the case that a file called FILE.INC or FILE.P also exists in the current directory.

From that point on, the native functions from the file I/O support library are available. Below is an example program that reads a (text) file and dumps the contents on the console:

LISTING: readfile.p

```
#include <file>

main()
    {
    /* ask for a filename */
    print "Please enter a filename: "
    new filename{128}
    getstring filename

    /* try to open the file */
    new File: file = fopen(filename, io_read)
    if (!file)
        {
       printf "The file '%s' cannot be opened for reading\n", filename
        exit
        }

    /* dump the file onto the console */
    new line{200}
    while (fread(file, line))
        print line, .highlight = true

    /* done */
    fclose file
    }
```

When you open a file for both reading and writing, you should call `fseek` when switching between reading and writing, to ensure that the disk caches are properly cleared.

Native functions

deletecfg

Deletes a key or a section from an INI file

Syntax:

```
bool: deletecfg(const filename[]="",
                const section[]="",
                const key[]="")
```

`filename` The name and path of the INI file. If this parameter is not set, the function uses the default name "`config.ini`".

`section` The section from which to delete the `key`. If this parameter is not set, the function stores the key/value pair outside any section.

`key` The key to delete. If this parameter is not set, the function deletes the entire section.

Returns: `true` on success, `false` on failure.

Notes: If parameters `section` and `key` are both not set, the function deletes all keys that are outside any sections.

See also: `readcfg`, `writecfg`

diskfree

Returns the free disk space

Syntax:

```
diskfree(const volume[]="")
```

`volume` The name of the volume on systems that support multiple disks or multiple memory cards. On single-volume systems, it is optional.

Returns: The amount of free space in KiB.

Notes: The maximum size that can be supported 2048 GiB (2 terabyte).

fattrib

Set the file attributes

Syntax:

```
bool: fattrib(const name[], timestamp=0,
              attrib=0x0f)
```

`name` The name of the file.

`timestamp` Time of the last modification of the file. When this parameter is set to zero, the time stamp of the file is not changed.

`attrib` A bit mask with the new attributes of the file. When set to 0x0f, the attributes of the file are not changed.

Returns: `true` on success and `false` on failure.

Notes: The time is in number of seconds since midnight at 1 January 1970: the start of the UNIX system epoch.

The file attributes are a bit mask. The meaning of each bit depends on the underlying file system (e.g. FAT, NTFS, etx2 and others).

See also: `fstat`

fblockread

Read an array from a file, without interpreting the data

Syntax:

```
fblockread(File: handle, buffer[],
           size=sizeof buffer)
```

`handle` The handle to an open file.

`buffer` The buffer to read the data into.

`size` The number of *cells* to read from the file. This value should not exceed the size of the `buffer` parameter.

Returns: The number of cells read from the file. This number may be zero if the end of file has been reached.

Notes: This function reads an array from the file, without encoding and ignoring line termination characters, i.e. in binary format. The number of bytes to read must be passed explicitly with the `size` parameter.

See also: `fblockwrite`, `fopen`, `fread`

fblockwrite

Write an array to a file, without interpreting the data

Syntax:
```
fblockwrite(File: handle, const buffer[],
            size=sizeof buffer)
```

`handle` The handle to an open file.

`buffer` The buffer that contains the data to write to the file.

`size` The number of *cells* to write to the file. This value should not exceed the size of the `buffer` parameter.

Returns: The number of cells written to the file.

Notes: This function writes an array to the file, without encoding, i.e. in binary format. The buffer need not be zero-terminated, and a zero cell does not indicate the end of the buffer.

See also: `fblockread`, `fopen`, `fwrite`

fclose

Close an open file

Syntax:
```
bool: fclose(File: handle)
```

`handle` The handle to an open file.

Returns: `true` on success and `false` on failure.

See also: `fopen`

fcopy

Copy a file

Syntax:
```
bool: fcopy(const source[], const target[])
```

`source` The name of the (existing) file that must be copied, optionally including a path.

`target` The name of the new file, optionally including a full path.

Returns: `true` on success and `false` on failure.

Notes: If the target file already exists, it is overwritten.

See also: `frename`

fexist

Count matching files, check file existence

Syntax: `fexist(const pattern[])`

`pattern` The name of the file, optionally containing wild-card characters.

Returns: The number of files that match the pattern

Notes: In the pattern, the characters "?" and "*" are wild-cards: "?" matches any character —but only exactly one character, and "*" matches zero or more characters. Only the final part of the path (the portion behind the last slash or backslash) may contain wild-cards.

If no wild-cards are present, the function returns 1 if the file exists and 0 if the file cannot be found. As such, you can use the function to verify whether a file exists.

Depending on the operating system or file system, the pattern matching may be case sensitive.

See also: `fmatch`

fgetchar

Read a single character (byte)

Syntax: `fgetchar(File: handle, bool: utf8=true)`

`handle` The handle to an open file.

`utf8` If the argument `utf8` is `true`, the function interprets UTF-8 encoding and may read multiple bytes from the file to form an extended character. If, on the other hand, the `utf8` argument is `false`, the function reads a single byte from the file and returns it as is.

Returns: The character that was read, or `EOF` on failure.

See also: `fopen`, `fputchar`

filecrc

Return the 32-bit CRC value of a file

Syntax: `filecrc(const name[])`

`name` The name of the file.

Returns: The 32-bit CRC value of the file, or zero if the file cannot be opened.

Notes: The CRC value is a useful measure to check whether the contents of a file has changed during transmission or whether it has been edited (provided that the CRC value of the original file was saved). The CRC value returned by this function is the same as the one used in ZIP archives (PKZip, WinZip) and the "SFV" utilities and file formats.

See also: `fstat`

flength

Return the length of an open file

Syntax: `flength(File: handle)`

`handle` The handle to an open file.

Returns: The length of the file, in bytes.

See also: `fopen`, `fstat`

fmatch

Find a filename matching a pattern

Syntax:
```
bool: fmatch(name[], const pattern[], index=0,
             maxlength=sizeof name)
```

`name` If the function is successful, this parameter will hold a n^{th} filename matching the pattern. The name is always returned as a packed string.

`pattern` The name of the file, optionally containing wild-card characters.

`index` The number of the file in case there are multiple files matching the pattern. Setting this parameter to 0 returns the first matching file, setting it to 1 returns the second matching file, etc.

`size` The maximum size of parameter `name` in cells.

Returns: `true` on success and `false` on failure.

Notes: In the pattern, the characters "?" and "*" are wildcards: "?" matches any character —but only exactly one character, and "*" matches zero or more characters. Only the final part of the path (the portion behind the last slash or backslash) may contain wildcards.

Depending on the operating system or file system, the pattern matching may be case sensitive.

See also: `fexist`

---

fmkdir — Create a directory

Syntax:
```
bool: fmkdir(const name[])
```

`name` The name of the directory to create, optionally including a full path.

Returns: `true` on success and `false` on failure.

Notes: To delete the directory again, use `fremove`. The directory must be empty before it can be removed.

See also: `fremove`

---

fopen — Open a file for reading or writing

Syntax:
```
File: fopen(const name[],
            filemode: mode=io_readwrite)
```

`name` The name of the file, including the path. It must adhere to the conventions of the operating system.

`mode` The intended operations on the file. It must be one of the following constants:

`io_read`
: opens an existing file for reading only (the file must already exist)

`io_write`
: creates a new file (or truncates an existing file) and opens it for writing only

`io_readwrite`
: opens a file for both reading and writing; if the file does not exist, a new file is created

`io_append`
: opens a file for writing only, where all (new) information is appended behind the existing contents of the file; if the file does not exist, a new file is created

Returns: A “handle” or “magic cookie” that refers to the file. If the return value is zero, the function failed to open the file.

See also: `fclose`

---

fputchar

Write a single character to the file

Syntax:
```
bool: fputchar(File: handle, value,
               bool: utf8=true)
```

`handle`
: The handle to an open file.

`value`
: The value to write (as a single character) to the file.

`utf8`
: If the argument `utf8` is `true`, the function writes the `value` in UTF-8 encoding, meaning that any value above 127 will be expanded into multiple bytes in the file. If the `utf8` argument is `false`, the function writes a single byte to the file; values above 255 are not supported.

Returns: `true` on success and `false` on failure.

See also: `fgetchar`, `fopen`

fread

Reads a line from a text file

Syntax:
```
fread(File: handle, string[],
      size=sizeof string, bool: pack=false)
```

`handle` The handle to an open file.

`string` The array to store the data in; this is assumed to be a text string.

`size` The (maximum) size of the array in cells. For a packed string, one cell holds multiple characters.

`pack` If the `pack` parameter is `false`, the text is stored as an *unpacked* string and the function parses UTF-8 encoding. When reading text in a *packed* string, no UTF-8 interpretation occurs.

Returns: The number of characters read. If the end of file is reached, the return value is zero.

Notes: Reads a line of text, terminated by CR, LF or CR–LF characters, from to the file. Any line termination characters are stored in the string.

See also: `fblockread`, `fopen`, `fwrite`

fremove

Delete a file or directory

Syntax:
```
bool: fremove(const name[])
```

`name` The name of the file or the directory.

Returns: `true` on success and `false` on failure.

Notes: A directory can only be removed if it is empty.

See also: `fmkdir`, `fexist`, `fopen`

frename

Rename a file

Syntax: `bool: frename(const oldname[], const newname[])`

`oldname` The current name of the file, optionally including a full path.

`newname` The new name of the file, optionally including a full path.

Returns: `true` on success and `false` on failure.

Notes: In addition to changing the name of the file, this function can also move the file to a different directory.

See also: `fcopy`, `fremove`

fseek

Set the current position in a file

Syntax:
```
fseek(File: handle, position=0,
      seek_whence: whence=seek_start)
```

`handle` The handle to an open file.

`position` The new position in the file, relative to the parameter `whence`.

`whence` The starting position to which parameter `position` relates. It must be one of the following:

`seek_start`
: Set the file position relative to the start of the file (the `position` parameter must be positive);

`seek_current`
: Set the file position relative to the current file position: the `position` parameter is added to the current position;

`seek_end`
: Set the file position relative to the end of the file (parameter `position` must be zero or negative).

Returns: The new position, relative to the start of the file.

Notes: You can either seek forward or backward through the file.

To get the current file position without changing it, set the `position` parameter to zero and `whence` to `seek_current`.

See also: `fopen`

---

fstat

Return the size and the time stamp of a file

Syntax:
```
bool: fstat(const name[], &size=0,
            &timestamp=0, &attrib=0, &inode=0)
```

`name`	The name of the file.
`size`	If the function is successful, this parameter holds the size of the file on return.
`timestamp`	If the function is successful, this parameter holds the time of the last modification of the file on return.
`attrib`	If the function is successful, this parameter holds the file attributes.
`inode`	If the function is successful, this parameter holds inode number of the file. An inode number is a number that uniquely identifies a file, and it usually indicates the physical position of (the start of) the file on the disk or memory card.

Returns: `true` on success and `false` on failure.

Notes: In contrast to the function `flength`, this function does not need the file to be opened for querying its size.

The time is in number of seconds since midnight at 1 January 1970: the start of the UNIX system epoch.

The file attributes are a bit mask. The meaning of each bit depends on the underlying file system (e.g. FAT, NTFS, etx2 and others).

See also: `fattrib`, `flength`

ftemp

Create a temporary file

Syntax: `File: ftemp()`

Returns: A handle to the temporary file (or zero on failure).

Notes: Creates a new temporary file, with a random or semi-random filename and usually in the "TMP" or "TEMP" subdirectory, for both reading and writing. When you close the file (`fclose`), it is automatically deleted.

See also: `fclose`, `fopen`

fwrite

Write a string to a file

Syntax: `fwrite(File: handle, const string[])`

`handle` The handle to an open file.

`string` The string to write to the file.

Returns: The number of characters actually written; this may be a different value from the string length in case of a writing failure ("disk full", or quota exceeded).

Notes: This function writes a zero-terminated buffer, presumably a text string, to the file. If the text string is in *unpacked* format, it is written to the file in UTF-8 encoding. A *packed* string is written to the file "as is".

The function does not append line-ending characters to the line of text written to the file (line ending characters are CR, LF or CR–LF characters).

See also: `fblockwrite`, `fopen`, `fread`

readcfg

Reads a text field from an INI file

Syntax:
```
readcfg(const filename[]="",
        const section[]="", const key[],
        value[], size=sizeof value,
        const defvalue[]="", bool: pack=false)
```

`filename` The name and path of the INI file. If this parameter is not set, the function uses the default name "`config.ini`".

`section` The section to look for the `key`. If this parameter is not set, the function reads the key outside any section.

`key` The key whose value must be looked up.

`value` The buffer into which to store the value. If the `key` is not present in the appropriate `section` of the INI file, the contents of parameter `defvalue` is copied into this buffer.

`size` The (maximum) size of the `value` array in cells. For a packed string, one cell holds multiple characters.

`defvalue` The string to copy into parameter `value` in case that the function cannot read the field from the INI file.

`pack` If the `pack` parameter is `false`, the text is stored as an *unpacked* string and the function parses UTF-8 encoding. When reading text in a *packed* string, no UTF-8 interpretation occurs.

Returns: The number of characters stored in parameter `value`.

See also: `readcfgvalue`, `writecfg`

---

readcfgvalue

Reads a numeric field from an INI file

Syntax:

```
readcfgvalue(const filename[]="",
             const section[]="", const key[],
             defvalue=0)
```

`filename` The name and path of the INI file. If this parameter is not set, the function uses the default name "`config.ini`".

`section` The section to look for the `key`. If this parameter is not set, the function reads the key outside any section.

`key` The key whose value must be looked up.

`defvalue` The value to return in case that the function cannot read the field from the INI file.

Returns: The numeric value if the field, or the value of `defvalue` if the field was not found in the section and/or at the key.

See also: `readcfg`, `writecfgvalue`

---

writecfg

Writes a text field to an INI file

Syntax:

```
bool: writecfg(const filename[]="",
               const section[]="",
               const key[], const value[])
```

`filename` The name and path of the INI file. If this parameter is not set, the function uses the default name "`config.ini`".

`section` The section to store the `key` under. If this parameter is not set, the function stores the key/value pair outside any section.

`key` The key for the field.

`value` The value for the field.

Returns: `true` on success, `false` on failure.

See also: `deletecfg`, `readcfg`, `writecfgvalue`

---

writecfgvalue

Writes a numeric field to an INI file

Syntax:

```
bool: writecfgvalue(const filename[]="",
                    const section[]="",
                    const key[], value)
```

`filename` The name and path of the INI file. If this parameter is not set, the function uses the default name "`config.ini`".

`section` The section to store the `key` under. If this parameter is not set, the function stores the key/value pair outside any section.

`key`	The key for the field.
`value`	The value for the field, as a signed (decimal) number.

Returns: `true` on success, `false` on failure.

See also: `readcfgvalue`, `writecfg`

Resources

The PAWN toolkit can be obtained from **www.compuphase.com** in various formats (binaries and source code archives). The manuals for usage of the language and implementation guides are also available on the site in Adobe Acrobat format (PDF files).

Documentation on Unicode and the Basic Multilingual Plane is found on **http://www.unicode.org**. A page for the UTF-8 encoding, **http://www.utf-8.com** contains a link to RFC 2279, and other information.

Index

◇ Names of persons (not products) are in *italics*.
◇ Function names, constants and compiler reserved words are in `typewriter font`.