Pawn

embedded scripting language

File I/O Support Library

March 2004

Abstract

The "File I/O Support Library" is an interface to standard files (text and binary) for the PAWN scripting language. The library provides simple read and write functions; it supports UTF-8 for encoding the Unicode character set.

The software that is associated with this application note can be obtained from the company homepage, see section "Resources"

Requests for corrections and additions to the manual and the software can be directed to ITB CompuPhase at the above address.

Typeset with TEX in the "Computer Modern" and "Palatino" typefaces at a base size of 11 points.

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 non-withstanding, 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), the 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 lines and blocks from a file, and for writing lines/blocks to a file. The line reading functions are for text files and the block reading functions for binary files. Additional functions allow you to read through a file character by character, or byte by byte, and to write a file character by character. The character reading/writing 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 and 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`),

deleting a file (`fremove`), creating a temporary file which is automatically deleted when you close it (`ftemp`), and browsing through the file (`fseek` and `flength`).

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/character encoding called UCS-4 and a 2-byte/character 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.

This file I/O support library for PAWN 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

* The actual name depends on how the library is implemented, see the chapter "Implementing the 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 filename `local/myfile.txt` refers to `/tmp/local/myfile.txt`. Prefixing the `local` subdirectory with a slash, to specify the `local` directory from the root, does not have any effect: the path still refers to `/tmp/local/myfile.txt`. 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 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 "Implementor'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 Implementor'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.PAWN 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 char]
    getstring filename, .pack=true

    /* 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

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

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.

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

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

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. The number of bytes to write must be passed explicitly with the `size` parameter.

The function returns the number of cells written to the file.

bool:fclose(File:handle)

Closes a file that was opened with `fopen`. The function returns `true` on success and `false` on failure.

bool:fexist(const name[])

Returns `true` or `false` depending on whether a file with the specified name and path exists.

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

Reads one character from the file. The functions returns the character read, or `EOF` on failure. 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.

flength(File:handle)

Returns the length of a file, in bytes.

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

Open the file, for reading and/or writing. The parameter `name` is the filename of the file to open; it must adhere to the conventions of the operating system.

The `mode` parameter is one of the following constants:

`io_read` opens and existing file for reading only;

`io_write` creates a new 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.

The function returns a "handle" or "magic cookie" that refers to the file. If the return value is zero, the function failed to open the file.

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

Writes one character to the file. The functions returns `true` on success and `false` on failure. 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.

fread(File:handle, string[], size = sizeof string, bool

pack=false): Reads a line of text, terminated by CR, LF or CR–LF characters, from to the file. If the `pack` parameter is `false`, the text is stored as an *unpacked* string and the function interprets UTF-8 encoding. When reading text in a *packed* string, no UTF-8 interpretation occurs. Note that a packed string can hold more characters than the number of cells in the `string` argument.

The function returns the number of characters read. If the end of file is reached, the return value can be zero.

Any line termination characters are stored in the string.

bool:fremove(const name[])

Removes, or deletes, the file with the given name. The function returns `true` on success and `false` on failure.

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

Sets the current position in the file and returns the new position. You can either seek forward or backward through the file.

The parameter `whence` should be one of the following:

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

`seek_current` Set the file position relative the current file position, i.e. the new file position is the current position plus the value of the `position` parameter;

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

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

File:ftemp()

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 with `fclose`, it is automatically deleted.

fwrite(File:handle, const string[])

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 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).

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).

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 (BMP) appears 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. To test the robustness of an UTF-8 decoder, the test file by Markus Kuhn is very valuable; see **http://www.cl.cam.ac.uk/ mgk25**.

Index

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