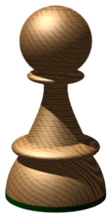
Pawn



embedded scripting language

String Processing

Contents

January 2015 CompuPhase

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. The "String Processing" module intends to be such a general-purpose library.

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.

Packed and unpacked strings

The PAWN language does not have variable types. All variables are "cells" which are typically 32-bit wide (there exist implementations of PAWN that use 64-bit cells). A string is basically an array of cells that holds characters and that is terminated with the special character '\0'.

However, in most character sets a character typically takes only a single byte and a cell typically is a four-byte entity: storing a single character per cell is then a 75% waste. For the sake of compactness, PAWN supports *packed* strings, where each cell holds as many characters as fit. In our example, one cell would contain four characters, and there is no space wasted.

At the same time, PAWN also supports *unpacked* strings where each cell holds only a single character, with the purpose of supporting Unicode or other wide-character sets. The Unicode character set is usually represented as a 16-bit character set holding the 60,000 characters of the Basic Multilingual Plane (BMP), and access to other "planes" through escape codes. A PAWN script can hold all characters of all planes in a cell, since a cell is typically at least 32-bit, without needing escape codes.

Many programming language solve handling of ASCII/Ansi character sets versus Unicode with their typing system. A function will then work either on one or on the other type of string, but the types cannot be mixed. PAWN, on the other hand, does not have types or a typing system, but it can check, at run time, whether a string a packed or unpacked. This also enables you to write a single function that operates on both packed and unpacked strings.

The functions in this String Processing library have been constructed so that they work on packed and unpacked strings.

UU-encoding

For transmitting binary data over communication lines/channels or protocols that do not support 8-bit transfers, or that reserve some byte values for special "control characters", a 6-bit data encoding scheme was devised that uses only the standard ASCII range. This encoding is called "UU-encoding".

This daemon can encode a stream of binary data into ASCII strings that can be transmitted over all networks that support ASCII.

The basic scheme is to break groups of 3 eight bit bytes (24 bits) into 4 six bit characters and then add 32 (a space) to each six bit character which maps it into the readily transmittable character. As some transmission mechanisms compress or remove spaces, spaces are changed into back-quote characters (ASCII 96) —this is a modification of the scheme that is not present in the original versions of the UU-encode algorithm.

Another way of phrasing this is to say that the encoded 6 bit characters are mapped into the set:

```
`!"#$%&'()*+,-./012356789:;<=>?@ABC...XYZ[\]^_
```

for transmission over communications lines.

A small number of eight bit bytes are encoded into a single line and a count is put at the start of the line. Most lines in an encoded file have 45 encoded bytes. When you look at a UU-encoded file note that most lines start with the letter "M". "M" is decimal 77 which, minus the 32 bias, is 45. The purpose of this further chopping of the byte stream is to allow for handshaking. Each chunk of 45 bytes (61 encoded characters, plus optionally a newline) is transferred individually and the remote host typically acknowledges the receipt of each chunk.

Some encode programs put a check character at the end of each line. The check is the sum of all the encoded characters, before adding the mapping, modulo 64. Some encode programs have bugs in this line check routine; some use alternative methods such as putting another line count character at the end of a line or always ending a line with an “M”. The functions in this module encode byte arrays without line check characters, and the decoder routine ignores any “check” characters behind the data stream.

To determine the end of a stream of UU-encoded data, there are two common conventions:

- ◇ When receiving a line with less that 45 encoded bytes, it signals the last line. If the last line contains 45 bytes exactly, another line with zero bytes must follow. A line with zero encoded bytes is a line with only a back-quote.
- ◇ A stream must always be ended with a line with 0 (zero) encoded bytes. Receiving a line with less than 45 encoded bytes does not signal the end of the stream — it may indicate that further data is only delayed.

Implementing the library

The implementation of the "String Processing" module is in the files AMXSTRING.C and STRING.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 AMXSTRING.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_StringInit`. The "clean-up" function is `amx_StringCleanup`, but 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.

Usage

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

```
#include <string>
```

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

From that point on, the native functions from the string manipulation library are available.

Several functions have a parameter that specifies the maximum number of *cells* that a destination buffer can hold. The purpose of this parameter is to avoid an accidental buffer overrun. Note that this parameter *always* gives the buffer size in *cells*, even for packed strings. The rationale behind this choice is that the `sizeof` operator of PAWN also returns the size of buffers in cells.

Native functions

ispacked

Determine whether a string is packed or unpacked

Syntax:	`bool: ispacked(const string[])`
	`string` The string to verify the packed/unpacked status for.
Returns:	`true` if the parameter refers to a packed string, and `false` otherwise.

memcpy

Copy bytes from one location to another

Syntax:

```
memcpy(dest[], const source[], index=0,
           numbytes, maxlength=sizeof dest)
```

`dest`	An array where the bytes from `source` are copied in.
`source`	The source array.
`index`	The index, in *bytes* in the source array starting from which the data should be copied.
`numbytes`	The number of bytes (not cells) to copy.
`maxlength`	The maximum number of *cells* that fit in the destination buffer.

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

Notes: This function can align byte strings in cell arrays, or concatenate two byte strings in two arrays. The parameter `index` is a byte offset and `numbytes` is the number of bytes to copy.

This function allows copying in-place, for aligning a byte region inside a cell array.

See also: `strcopy`, `strpack`, `strunpack`, `uudecode`, `uuencode`

strcat

Concatenate two strings

Syntax:
```
strcat(dest[], const source[],
       maxlength=sizeof dest)
```

`dest`	The buffer holding the initial string on entry and the resulting string on return.
`source`	The string to append to the string in parameter `dest`.
`maxlength`	The size of `dest` in cells. If the length of `dest` would exceed `maxlength` cells after the string concatenation, the result is truncated to `maxlength` cells.

Returns: The string length of `dest` after concatenation.

Notes: During concatenation, the `source` string may be converted from packed to unpacked, or vice versa, in order to match `dest`. If `dest` is an empty string, the function makes a plain copy of `source`, meaning that the result (in `dest`) will be a packed string if `source` is packed too, and unpacked otherwise.

See also: `strcopy`, `strins`, `strpack`, `strunpack`

strcmp

Compare two strings

Syntax:
```
strcmp(const string1[], const string2[],
       bool: ignorecase=false, length=cellmax)
```

`string1`	The first string in the comparison.
`string2`	The first string in the comparison.
`ignorecase`	If logically “true”, case is ignored during the comparison.
`length`	The maximum number of characters to consider for comparison.

Returns: The return value is:
-1 if `string1` comes *before* `string2`,
1 if `string1` comes *after* `string2`, or
0 if the strings are equal (for the matched length).

Notes: Packed and unpacked strings may be mixed in the comparison.

This function does *not* take the sort order of non-ASCII character sets into account. That is, no Unicode "Collation Algorithm" is used.

See also: `strequal`, `strfind`

---

strcopy

Create a copy of a string

Syntax:

```
strcopy(dest[], const source[],
        maxlength=sizeof dest)
```

`dest`	The buffer to store the copy of the string string in.
`source`	The string to copy, this may be a packed or an unpacked string.
`maxlength`	The size of `dest` in cells. If the length of `dest` would exceed `maxlength` cells, the result is truncated. Note that a cell can hold multiple packed characters.

Returns: The number of characters copied.

Notes: This function copies a string from `source` to `dest`. If the source string is a packed string, the destination will be packed too; likewise, if the source string is unpacked, the destination will be unpacked too. See functions `strpack` and `strunpack` to convert between packed and unpacked strings.

See also: `strcat`, `strpack`, `strunpack`

---

strdel

Delete characters from the string

Syntax:

```
bool: strdel(string[], start, end)
```

`string`	The string from which to remove a range characters.
`start`	The index of the first character to remove (starting at zero).

`end`	The parameter `end` must point *behind* the last character to remove.

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

Notes: For example, to remove the letters “ber” from the string “Jabberwocky”, set `start` to 3 and `end` to 6.

See also: `strins`

strequal

Compare two strings

Syntax:

```
bool: strequal(const string1[], const
                    string2[],
                    bool: ignorecase=false,
                    length=cellmax)
```

`string1`	The first string in the comparison.
`string2`	The first string in the comparison.
`ignorecase`	If logically “true”, case is ignored during the comparison.
`length`	The maximum number of characters to consider for

Returns: `true` if the strings are equal, `false` if they are different.

See also: `strcmp`

strfind

Search for a sub-string in a string

Syntax:

```
strfind(const string[], const sub[],
        bool: ignorecase=false, index=0)
```

`string`	The string in which you wish to search for sub-strings.
`sub`	The sub-string to search for.
`ignorecase`	If logically “true”, case is ignored during the comparison.
`index`	The character position in `string` to start searching. Set to 0 to start from the beginning of the string.

Returns: The function returns the character index of the first occurrence of the string `sub` in `string`, or -1 if no occurrence was found. If an occurrence was found, you can search for the next occurrence by calling `strfind` again and set the parameter `offset` to the returned value plus one.

Notes: This function searches for a sub-string in a string, optionally ignoring the character case and optionally starting at an offset in the string.

See also: `strcmp`

---

strformat

Convert values to text

Syntax:

```
strformat(dest[], size=sizeof dest,
          bool: pack=false, const format[],
          ...)
```

`dest`	The string that will contain the formatted result.
`size`	The maximum number of *cells* that the `dest` parameter can hold. This value includes the zero terminator.
`pack`	If `true`, the string in `dest` will become a packed string. Otherwise, the string in `dest` will be unpacked.
`format`	The string to store in `dest`, which may contain placeholders (see the notes below).
`...`	The parameters for the placeholders. These values may be untagged, weakly tagged, or tagged as rational values.

Returns: This function always returns 0.

Notes: The `format` parameter is a string that may contain embedded *placeholder* codes:

`%c`	store a character at this position
`%d`	store a number at this position in decimal radix
`%f`	store a floating point number at this position (for implementations that support floating point)
`%q`	store a fixed point number at this position

`%r`	same as either `%f` or `%q` (for compatibility with other implementations of PAWN)
`%s`	store a character string at this position
`%x`	store a number at this position in hexadecimal radix

The values for the placeholders follow as parameters in the call.

You may optionally put a number between the "`%`" and the letter of the placeholder code. This number indicates the field width; if the size of the parameter to print at the position of the placeholder is smaller than the field width, the field is expanded with spaces.

The `strformat` function works similarly to the "C" function `sprintf`.

See also: `valstr`

---

strins

Insert a sub-string in a string

Syntax:

```
bool: strins(string[], const substr[], index,
                    maxlength=sizeof string)
```

`string`	The source and destination string.
`substr`	The string to insert in parameter `string`.
`index`	The character position of `string` where `substr` is inserted. When 0, `substr` is prepended to `string`.
`maxlength`	The size of `dest` in cells. If the length of `dest` would exceed `maxlength` cells after insertion, the result is truncated.

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

Notes: During insertion, the `substr` parameter may be converted from a packed string to an unpacked string, or vice versa, in order to match `string`.

If the total length of `string` would exceed `maxlength` cells after inserting `substr`, the function raises an error.

See also: `strcat`, `strdel`

strlen

Return the length of a string

Syntax: `strlen(const string[])`

`string`	The string to get the length from.

Returns: The length of the string in characters (not the number of cells). The string length *excludes* the terminating "\0" character.

Notes: Like all functions in this library, the function handles both packed and unpacked strings.

To get the number of *cells* held by a packed string of a given length, you can use the predefined constants `charbits` and `cellbits`.

See also: `ispacked`

strmid

Extract a range of characters from a string

Syntax:
```
strmid(dest[], const source[],
       start=0, end=cellmax,
       maxlength=sizeof dest)
```

`dest`	The string to store the extracted characters in.
`source`	The string from which to extract characters.
`start`	The index of the first character to extract (starting at zero).
`end`	The index of the character *after/* the last character to extract.
`maxlength`	The size of `dest` in cells. If the length of `dest` would exceed `maxlength` cells, the result is truncated.

Returns: The number of characters stored in `dest`.

Notes: The parameter `start` must point at the first character to extract (starting at zero) and the parameter end must point *behind* the last character to extract. For example, when the source string contains "Jabberwocky", `start` is 1 and `end` is 5, parameter `dest` will contain "abbe" upon return.

See also: `strdel`

strpack

Create a "packed" copy of a string

Syntax:

```
strpack(dest[], const source[],
        maxlength=sizeof dest)
```

`dest`	The buffer to store the packed string in.
`source`	The string to copy, this may be a packed or an unpacked string.
`maxlength`	The size of `dest` in cells. If the length of `dest` would exceed `maxlength` cells, the result is truncated. Note that a cell may hold multiple packed characters.

Returns: The number of characters copied.

Notes: This function copies a string from `source` to `dest` and stores the destination string in packed format. The source string may either be a packed or an unpacked string.

See also: `strcat`, `strunpack`

strunpack

Create an "unpacked" copy of a string

Syntax:

```
strunpack(dest[], const source[],
          maxlength=sizeof dest)
```

`dest`	The buffer to store the unpacked string in.
`source`	The string to copy, this may be a packed or an unpacked string.
`maxlength`	The size of `dest` in cells. If the length of `dest` would exceed `maxlength` cells, the result is truncated.

Returns: The number of characters copied.

Notes: This function copies a string from `source` to `dest` and stores the destination string in unpacked format.

The source string may either be a packed or an unpacked string.

See also: `strcat`, `strpack`

strval

Convert from text (string) to numbers

Syntax: `strval(const string[], index=0)`

`string`	The string containing a number in characters. This may be either a packed or unpacked string.
`index`	The position in the string where to start looking for a number. This parameter allows to skip an initial part of a string, and extract numbers from the middle of a string.

Returns: The value in the string, or zero if the string did not start with a valid number (starting at `index`).

See also: `valstr`

uudecode

Decode an UU-encoded stream

Syntax:
```
uudecode(dest[], const source[],
         maxlength=sizeof dest)
```

`dest`	The array that will hold the decoded byte array.
`source`	The UU-encoded source string.
`maxlength`	The size of `dest` in cells. If the length of `dest` would exceed `maxlength` cells, the result is truncated. Note that multiple bytes fit in each cell.

Returns: The number of *bytes* decoded and stored in `dest`.

Notes: Since the UU-encoding scheme is for binary data, the decoded data is always "packed". The data is unlikely to be a string (the zero-terminator may not be present, or it may be in the middle of the data).

A buffer may be decoded "in-place"; the destination size is always smaller than the source size. Endian issues (for multi-byte values in the data stream) are not handled.

Binary data is encoded in chunks of 45 bytes. To assemble these chunks into a complete stream, function `memcpy` allows you to concatenate buffers at byte-aligned boundaries.

See also: `memcpy`, `uuencode`

uuencode

Encode an UU-encoded stream

Syntax:
```
uuencode(dest[], const source[], numbytes,
         maxlength=sizeof dest)
```

`dest`	The buffer that will hold the UU-encoded string.
`source`	The byte array.
`numbytes`	The number of bytes (in the `source` array) to encode. This should not exceed 45.
`maxlength`	The size of `dest` in cells.

Returns: Returns the number of characters encoded, excluding the zero string terminator; if the dest buffer is too small, not all bytes are stored.

Notes: This function always creates a packed string. The string has a newline character at the end.

Binary data is encoded in chunks of 45 bytes. To extract 45 bytes from an array with data, possibly from a byte-aligned address, you can use the function `memcpy`.

A buffer may be encoded "in-place" if the destination buffer is large enough. Endian issues (for multi-byte values in the data stream) are not handled.

See also: `memcpy`, `uudecode`

valstr

Convert a number to text (string)

Syntax: `valstr(dest[], value, bool: pack=false)`

`dest`	The string to store the text representation of the number in.
`value`	The number to put in the string `dest`.
`pack`	If `true`, `dest` will become a packed string, otherwise it will be an unpacked string.

Returns: The number of characters stored in `dest`, excluding the terminating “\0” character.

Notes: Parameter `dest` should be of sufficient size to hold the converted number. The function does not check this.

See also: `strval`

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**.

Index

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