Pawn

embedded scripting language

Script Arguments Support Library

August 2006

Abstract

The "Script Arguments Support Library" provides functions to general purpose command line parsing to a PAWN script. The "command line" options may either come from the host program's actual command line (the default), or from a pseudo command line that the host program provides to the scripts.

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

Introduction 1
Command line syntax 1
Implementing the library 2
Usage 4
Native functions 5
Resources 7
Index 9

ITB CompuPhase

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

Introduction

The Script Arguments Support Library enables scripts or PAWN programs to accept options from a general purpose "options line" that the host program supplies, or from the command line of the host program itself. Using this support library, the host program may allow PAWN scripts to browse through the options with which the host program itself was started and to select and process a subset of these options. This makes it especially convenient for a host application to forward command line options to a script of which it has no intrinsic support itself. The host program may also choose an alternative (and specific) way to pass options to a script.

This appendix 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 site **www.compuphase.com**.

Command line syntax

The Script Arguments Support Library supports command lines where options are separated with white space (space characters, tab characters). The library does not mandate a specific option character. On some platforms, the default command line may have limited length —for example, on DOS the command line is limited to 127 characters. A host application may overcome this limitation by calling the function `amx_ArgsSetCmdLine`, see page 2.

When options contain space characters, they must be quoted. To quote an option, enclose the entire option between double quotes. For example, to pass an option with the name `-prompt` and the value `hello world`, the option would be:

```
  "-prompt=hello world"
```

When searching through the command line for options, the functions split an optional value for the option from the name. In the above example, the name of the option is "`-prompt`" and the value is "`hello world`". The separator between the name and the value of the option is an equal sign ("="). On UNIX/Linux platforms, the colon (":") may also be used to separate values from names. On Microsoft Windows and DOS, the colon is disabled (as a separator) by default, but a host application may enable it by (re-)building the library —see the option `AMXARGS_COLON` on page 2.

Implementing the library

The Script Arguments Support library consists of this document and the files AMXARGS.C and ARGS.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 AMXARGS.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_ArgsInit` and the "clean-up" function is `amx_ArgsCleanup`. 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 allows some configuration through preprocessor macros that can be specified as compiler/build options:

- ◇ The preprocessor macro `AMXARGS_COLON` may be set to 0 or 1 (zero or one); when 1, argument values may be separated from the option name with a colon (as well as with an equal sign). By default, colon separators are allowed on Linux, but not on Microsoft Windows and DOS. On Windows and DOS, a colon is also used in path names (the "drive" identifier), so using it as a name/value separator in an option may not be advisable.
- ◇ The preprocessor macro `AMXARGS_SKIPARG` may be set to 0 or 1 (zero or one) to optionally skip the first option on the "command line". The first option may be the name of the PAWN script (this is common if the host application takes the name of the script as the first parameter). By default, the first option is *not* ignored.

The default operation of the extension module is that it retrieves the command line options of the host program, so that the script can look up options that have a meaning for the script. A host program may also choose to build up a command line for the script with different options than those on the command line. In this case, the host program must pass the argument list to the extension module via function `amx_ArgsSetCmdLine`. The prototype for the function is:

```
int amx_ArgsSetCmdLine(const TCHAR *cmdline);
```

The function returns zero on success and an error code on failure. The parameter `cmdline` is a pointer to either a "`char`" string or a "`wchar_t`" string, depending on whether the library was compiled with Unicode support. Unicode support is only available in the Microsoft Windows build. The host application *must not delete* this buffer until the script(s) using the command line interface have all finished —the extension module *does not make a copy of the input parameter.*

Usage

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

```
#include <args>
```

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

From that point on, the native functions from the script argument support library are available. Below is an example program that prints all options passed to it on the console:

Listing: **argument.p**

```
#include <args>

main()
    {
    printf "Argument count = %d\n", argcount()

    new opt{100}
    for (new index = 0; argindex(index, opt); index++)
        printf "Argument %d = %s\n", index, opt
    }
```

The Script Arguments Support library supports, in its default configuration, the options on the command line of the host application. In this context, there will probably be a mix of options for the script and options for the host application itself on the command line. That is, not every option on the command line makes sense for the script. Rather than browsing through all options on the command line, it may be more convenient to "search" for any/all applicable options.

The library provides two native functions that allow to search for options. Both return `true` if the option exists and `false` if the option does not occur. If the option consists of a name/value pair, the functions split the value off the name and return it in a separate parameter. The difference between the two functions is that one of the two returns the "value" part as a text string and the other returns it as a numeric value. See the descriptions for `argstr` and `argvalue` for details.

Native functions

argcount()

Returns the number of arguments on the command line.

bool:argindex(*index*, *value*[], *maxlength*=sizeof value, bool:*pack*=true**)**

Looks up the indexed argument and stores it (if found) in the parameter `value`. The parameter `index` starts at zero for the first argument. The argument can be stored as a packed or an unpacked string; the parameter `pack` indicates whether packing is enabled.

Returns `true` on success and `false` on failure or if the parameter `index` is out of range.

bool:argstr(*index*=0, const *option*[]=`""`, *value*[]=`""`, *maxlength*=sizeof value, bool:*pack*=true**)**

Looks up a named option and splits of a value of the option, if any. The parameter `index` must be zero (0) to find the first occurrence of the option; when set to one (1) or higher to find for the second, third, ... occurrences of the named option. When the parameter `option` matches multiple arguments on the command line, the `index` parameter, then, enables to browse through them.

If a command line argument contains a "=" or a ":", the part to the left of the option is matched —it must include any "option character", such as a "-" or "/"; otherwise the entire argument is matched against the name "`option`". If `option` is an empty string, it will only match arguments without an "=" (or ":").

The parameter `value` is *not modified* if the option is not found. If the option was found, but contained no value, parameter `value` is set to an empty string.

Returns `true` if the option was found and `false` otherwise.

bool:argvalue(*index*=0, const *option*[]=`""`, &*value*=cellmin**)**

Looks up a named option and splits of a value of the option, if any. The parameter `index` must be zero (0) to find the first occurrence of the option; when set to one (1) or higher to find for the second, third, ... occurrences of the named option. When the parameter `option` matches multiple arguments on the command line, the `index` parameter, then, enables to browse through them.

If a command line argument contains a "=" or a ":", the part to the left of the option is matched —it must include any "option character", such as a "`-`" or "/"; otherwise the entire argument is matched against the name "`option`". If `option` is an empty string, it will only match arguments without an "=" (or ":").

The parameter `value` is *not modified* if the option is not found. If the option was found, but contained no value or no numeric value, parameter `value` is not changed from the input —this allows you to store a default value into the output parameter (parameter `value`) before the call.

Returns `true` if the option was found and `false` otherwise.

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).

Index

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

!

`#include`, 2

A

Abstract Machine, 2
Adobe Acrobat, 7
`argcount`, 5
`argindex`, 5
`argstr`, 5
`argvalue`, 5

D

DLL, 2

H

Host application, 2

L

Linux, 1, 2

M

Microsoft Windows, 1, 2

N

Native functions, 2
  registering, 2

P

Prefix file, 2
Preprocessor directive, 2

R

Registering, 2

S

Shared library, 2

U

Unicode, 3