# Pawn

embedded scripting language

## Process control & Foreign Function Interface

## Contents




January 2015 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



Typeset with TEX in the "DejaVu" typeface family.

# Introduction

The Process control and Foreign Function Interface module enables PAWN programs to run executable programs and call functions from libraries. When running executable programs, the PAWN script can also interact with that program (send "input", receive "output").

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**.

## Library calls

A PAWN script calls functions that are implemented inside the script itself and it calls functions that have been specifically registered for use from PAWN. These are PAWN functions and *native* functions, respectively. A Foreign Function Interface allows the PAWN script to call functions in general purpose libraries that do not adhere to any "PAWN" calling convention. Modern operating systems provide a large set of "system calls" that reside in pre-compiled libraries —in the Microsoft Windows world, these are DLLs, or Dynamically Linked Libraries; for UNIX, Linux and similar operating systems, these are *shared libraries*.

The Foreign Function Interface, therefore, makes this wealth of functions available to a PAWN script, but there are several caveats:

- Since the "foreign" functions are based on a different machine model than PAWN, you have to pass additional "bookkeeping" information to the function call.
- Calling functions in DLLs and shared libraries breaks the *sandbox* model of the PAWN abstract machine. It is very easy to crash an application by sending invalid parameters to a foreign function.
- And there are security & stability issues; see page 2

# Launching programs

The extension module supports launching executable programs. If the launched program is a “console application”, it can also provide input to the launched program and read its output. The external program sees the input as standard console/terminal input, and any text that it writes to standard console/terminal output is read back by the PAWN script.

In the current version, I/O redirection only works for a single launched program.

# Security, stability

The ability launch any program, or to call any function from a system DLL or shared library may obviously cause the system to become unstable. Passing the wrong parameters to a program or library function may crash the system immediately, and it could even damage (system) files.

When the PAWN abstract machine is implemented in an environment where scripts are automatically loaded from a network connection, it also becomes vulnerable to sabotage or “malware”.

# Implementing the library

The Process control and Foreign Function Interface module consists of this document and the file AMXPROCESS.C. This 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 PROCESS.INC file contains the definitions for the PAWN compiler of the native functions in AMXPROCESS.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_ProcessInit`. The current implementation provides "clean-up" function `amx_ProcessCleanup`, but using it 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.

When compiling the library under UNIX or Linux, you need the "`dyncall`" library, see `http://www.dyncall.org/`. When compiling for Microsoft Windows, you may also need this library, depending on your compiler. If the `dyncall` library is absent, the Process control module will still compile, but it will exclude the support for shared libraries.

# Usage

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

```
#include <process>
```

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

# Native functions

## libcall

Call a function in a library

Syntax:

```
libcall(const libname[], const funcname[],
        const typestring[], ...)
```

`libname` The module name or filename of the DLL or of the shared library. This library is automatically loaded on first use. The library name look-up is case-sensitive.

`funcname` The name of the function to call. This must be the symbol name as it exists in the library, including any "decorations" that are added by compilers or linkers.

`typestring` A string that tells the types of the function parameters, as defined by the foreign function. See the notes for the syntax.

`...` The values of the foreign parameters.

Returns: This function returns the return value of the foreign function.

Notes: The `typestring` format is a string where the types of the parameters that follow on the parameter list are represented by tokens. A token is basically a single letter, but it may be decorated with attributes. White space is permitted between the tokens, but not inside each token specification. The string "`ii[4]&u16s`" is equivalent to "`i i[4] &u16 s`" (but it is easier on the eye).

**Basic types**:

- `i` signed integer, by default 16-bit in Windows 3.x and 32-bit in Win32 and Linux
- `u` unsigned integer, by default 16-bit in Windows 3.x and 32-bit in Win32 and Linux
- `f` IEEE floating point, 32-bit (single precision)
- `p` packed string
- `s` unpacked string

Whether a string is *packed* or *unpacked*, is only relevant when the parameter is passed by reference (see below).

**Pass-by-value and pass-by-reference**:
By default, parameters are passed by value. To pass a parameter by reference, prefix the type letter with an "`&`". For example:

`&i` a signed integer passed by reference
`i` a signed integer passed by value

The same applies for "`&u`" versus "`u`" and "`&f`" versus "`f`".

Arrays are passed by "copy & copy-back". That is, `libcall` allocates a block of dynamic memory to copy the array into. On return from the foreign function, `libcall` copies the array back to the abstract machine. The net effect is similar to pass by reference, but the foreign function does not work in the AMX stack directly. During the copy and the copy-back operations, `libcall` may also transform the array elements, for example between 16-bit and 32-bit elements. This is done because PAWN only supports a single cell size, which may not fit the required integer size of the foreign function.

See "element ranges" (below) for the syntax of passing an array.

Strings may either be passed by copy, or by "copy & copy-back". When the string is an output parameter (for the foreign function), the size of the array that will hold the return string must be indicated between square brackets behind the type letter (see "element ranges", below). When the string is *input only*, this is not needed –`libcall` will determine the length of the input string itself.

The tokens "`p`" and "`s`" are equivalent, but "`p[10]`" and "`s[10]`" are not equivalent: the latter syntaxes determine whether the output from the foreign function will be stored as a packed or an unpacked string.

**Element sizes:**
To set the size in bits of a parameter, add an integer behind the type letter; for example, "`i16`" indicates

a 16-bit signed integer. Note that the value behind the type letter must be either 8, 16 or 32.

You should only use element size specifiers on the "`i`" and "`u`" types. That is, do not use these specifiers on "`f`", "`s`" and "`p`".

**Element ranges:**
For passing arrays, the size of the array should be given behind the type letter (and the optional element size). The token "`u[4]`" indicates an array of four unsigned integers, which are typically 32-bit. The token "`i16[8]`" is an array of 8 signed 16-bit integers. Arrays are always passed by "copy & copy-back".

When compiled as Unicode, this library converts all strings to Unicode strings, prior to calling the foreign function. Unicode functions should only be used with unpacked strings, since packed strings can only represent 8-bit character sets.

The calling convention for the foreign functions is assumed to be:
◇ `__stdcall` for Win32,
◇ `far pascal` for Win16
◇ and the GCC default for UNIX/Linux (`_cdecl`)

C++ name mangling of the called function is not handled (because C++ name mangling is not standardized and not portable). Win32 name mangling (used by default by Microsoft compilers on functions declared as `__stdcall`) is also not handled.

See also: `libfree`

---

## libfree

Unload a library from memory

Syntax: `bool: libfree(const libname[]="")`

`libname` The module name or filename of the DLL or of the shared library to remove from memory. If this parameter is an empty string (the default), all libraries that have been loaded by the extension module are removed.

Returns: `true` on success, `false` on failure (the function fails on an attempt to unload a DLL/shared library that was not loaded).

Notes: Function `libcall` keeps DLLs or shared libraries in memory after the first call to a function in it. This speeds up the subsequent calls to any of the functions in the same library, but it also increases the memory footprint of the process. Use `libfree` to unload DLLs or shared libraries that the script will not be needing again for some time. Doing so reduces the memory footprint of the process. At the next call to `libcall`, any unloaded library is automatically loaded again.

See also: `libcall`

---

## procexec — Launch an external executable program

Syntax:
```
PID: procexec(const commandline[])
```

`commandline`
: The filename or complete path of the external program, including any command line options.

Returns: An identifier for the new process ("Process ID", or *PID*) on success, or zero on failure.

See also: `procread`, `procwait`, `procwrite`

---

## procread — Read output of the external program

Syntax:
```
bool: procread(line[], size=sizeof line,
               bool: striplf=false,
               bool: packed=false)
```

`line`
: This parameter will contain the output of the external program (the text that the external program wrote to the console or "standard out").

`size`
: The size of parameter `line` parameter, in cells.

`striplf` If true, terminating newline and carriage return characters are stripped from parameter `line`.

`packed` If `true`, the returned text is stored in parameter `line` as a packed string; otherwise the string is unpacked.

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

See also: `procexec`, `procwrite`

---

## procwait

Wait until an executable program finishes

Syntax:
```
PID: procwait(PID: pid)
```

`pid` The Process ID of the program on which you wish to wait. This value is returned by `procexec`.

Returns: This function currently always returns zero.

See also: `procexec`

---

## procwrite

Send “input” to the external program

Syntax:
```
bool: procwrite(const line[],
                bool: appendlf=false)
```

`line` The text to send to the external program.

`appendlf` If `true`, a newline character is automatically appended to text that is sent.

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

See also: `procexec`, `procread`

# 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).

The `dyncall` library is available at `http://www.dyncall.org/`.

# Index

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