# Pawn

**embedded scripting language**

# Time Functions Library

**June 2005**

## Abstract

The "Time Functions Library" adds a set of general purpose functions to the PAWN scripting language. The library provides an interface to standard "time of the day" as well as a millisecond-resolution timer.

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

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. The "Time Functions Library" discussed in this document intends to be such a general-purpose module.

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.

# Implementing the library

The "Time Functions Library" consists of the two files AMXTIME.C and TIME.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 AMXTIME.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_TimeInit` and the "clean-up" function is `amx_TimeCleanup`. 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.

# Usage

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

```
#include <time>
```

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

From that point on, the native functions from the file I/O support library are available.

The `settimer` function sets up the interval (or delay) for the `@timer` callback function. To get a time event, the script must implement the `@timer` callback function and configure the timer with `settimer`.

An event-driven program that prints a period ("`.`") every second is:

Listing: **event-driven program to print a dot each second**

```
#include <time>

main()
    settimer 1000       /* interval is in milliseconds */

@timer()
    print "."
```

For comparison, below is a lineal program that does the same thing. It needs two loops: an inner loop to check for overflowing a second and an outer loop to continue printing dots after each second lapse. The program below is designed for purpose of demonstration, instead of timing quality. As it is, it is prone to timer drift. The event-driven alternative above is more accurate.

Listing: **lineal program to print a dot each second**

```
#include <time>

main()
    {
    for ( ;; )
        {
        new stamp = tickcount()
        while (tickcount() - stamp < 1000)
            {}
        print "."
        }
    }
```

# Public functions

**@timer** — A timer event occurred

Syntax: `@timer()`

Returns: The return value of this function is currently ignored.

Notes: This function executes after the delay set with `settimer`. Depending on the timing precision of the host, the call may occur later than the delay that was set.

If the timer was set as a "single-shot", it must be explicitly set again for a next execution for the `@timer` function. If the timer is set to be repetitive, `@timer` will continue to be called with the set interval until it is disabled with another call to `settimer`.

See also: `settimer`

# Native functions

**getdate** — Return the current (local) date

Syntax: `getdate(&year=0, &month=0, &day=0)`

| | |
|---|---|
| `year` | This will hold the year upon return. |
| `month` | This will hold the month (1–12) upon return. |
| `day` | This will hold the day of (1–31) the month upon return. |

Returns: The return value is the number of days since the start of the year. January 1 is day 1 of the year.

See also: `gettime`, `setdate`.

**gettime** — Return the current (local) time

Syntax: `gettime(&hour=0, &minute=0, &second=0)`

| | |
|---|---|
| `hour` | This will hold the hour (0–23) upon return. |
| `minute` | This will hold the minute (0–59) upon return. |
| `second` | This will hold the second (0–59) upon return. |

Returns: The return value is the number of seconds since 1 January 1970: the start of the Unix system era.

See also: `getdate`, `settime`.

**setdate** — Set the system date

Syntax: `setdate(year=0, month=0, day=0)`

| | |
|---|---|
| `year` | The year to set; if this parameter is zero, it is ignored. |
| `month` | The month to set; if this parameter is zero, it is ignored. |

`day` The month to set; if this parameter is zero, it is ignored.

Returns: This function always returns 0.

See also: `getdate`, `settime`.

---

**settime** Set the system time

Syntax: `settime(hour=-1, minute=-1, second=-1)`

The hour to set; if this parameter is negative, it is ignored.

The minute to set; if this parameter is negative, it is ignored.

The second to set; if this parameter is negative, it is ignored.

Returns: This function always returns 0.

See also: `gettime`, `setdate`.

---

**settimer** Configure the event timer

Syntax: `settimer(milliseconds, bool: singleshot=false)`

`milliseconds`
The number of milliseconds to wait before calling the `@timer` callback function. Of the timer is repetitive, this is the interval. When this parameter is 0 (zero), the timer is shut off.

`singleshot` If `false`, the timer is a repetitive timer; if `true` the timer is shut off after invoking the `@timer` event once.

Returns: This function always returns 0.

Notes: See the chapter "Usage" for an example of this function, and the `@timer` event function.

See also: `@timer`, `tickcount`.

**tickcount** Return the current tick count

Syntax: `tickcount(&granularity=0)`

`granularity` Upon return, this value contains the number of ticks that the internal system time will tick per second. This value therefore indicates the accuracy of the return value of this function.

Returns: The number of milliseconds since start-up of the system. For a 32-bit cell, this count overflows after approximately 24 days of continuous operation.

Notes: If the granularity of the system timer is "100" (a typical value for Unix systems), the return value will still be in milliseconds, but the value will change only every 10 milliseconds (100 "ticks" per second is 10 milliseconds per tick).

This function will return the time stamp regardless of whether a timer was set up with `settimer`.

See also: `settimer`.

# 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`.