# Pawn

embedded scripting language

# Implementer's Guide

# Table of contents

# Introduction

"PAWN" is a simple, typeless, 32-bit extension language with a C-like syntax. The language and features are described in the companion booklet with the sub-title "The Language". This "Implementer's Guide" discusses how to embed the PAWN scripting language in a host application.

The PAWN toolkit consists of two major parts: the compiler takes a script and converts it to P-code (or "bytecode"), which is subsequently executed on an abstract machine (or "virtual machine"). PAWN itself is written mostly in the C programming language (there are a few files in assembler) and it has been ported to Microsoft Windows, Linux, PlayStation 2 and the XBox. When embedding PAWN in host applications that are not written in C or C$^{++}$, I suggest that you use the AMX DLLs under Microsoft Windows.

There is a short chapter on the compiler. Most applications execute the compiler as a standalone utility with the appropriate options. Even when you link the compiler into the host program, its API is still based on options as if they were specified on the command line.

The abstract machine is a function library. The chapter devoted to it contains several examples for embedding the abstract machine in a host application, in addition to a reference to all API functions.

Appendices, finally, give compiling instructions for various platforms and background information —amongst others the debugger interface and the instruction set.

The PAWN language and toolset was designed to be an extension language for applications —as opposed to many other scripting languages that primarily aim at the command shell of the operating system. Being an extension language, the tools an libraries of the PAWN toolset must be *integrated* with the product.

The two main parts of the PAWN toolset are the compiler and the abstract machine. The compiler may either be linked into the host application, or it may be a separate process that is launched from the host application. For performance reasons, the abstract machine is always embedded (linked-in) inside the host application.
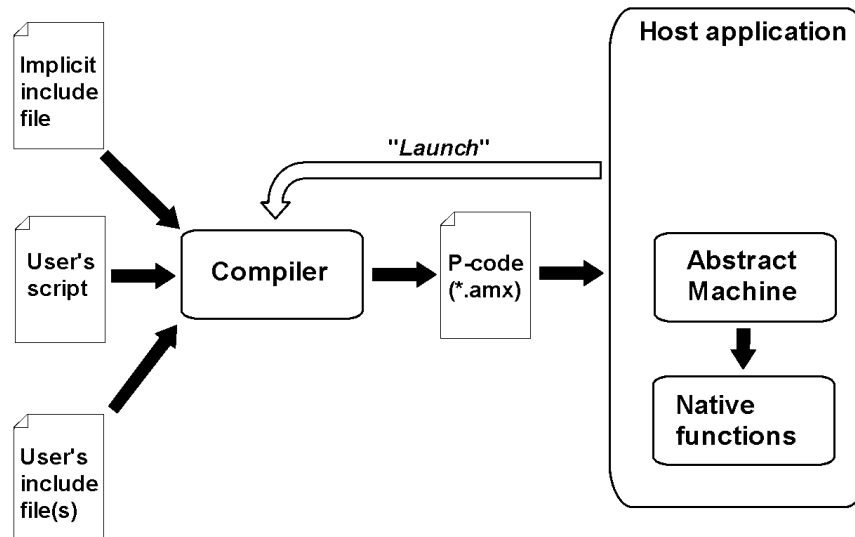
The PAWN compiler takes a series of text files containing the code for the user script and definitions of the environment/the host application. One of the include files is implicit: the PAWN compiler will automatically include it in any user script, but it will fail silently if that file is not present. The default name for that implicit include file (or "prefix file") is "DEFAULT.INC". You can override this name with a command line option to the PAWN compiler.

For a host application, it is advised to create an implicit include file containing:
◇ all "application specific" constants;
◇ all native functions that the host application provides (or a core subset of these native functions);
◇ all overloaded operators (or a core subset of these);
◇ all stock functions (or a core subset of these);
◇ forward declarations of all public functions;
◇ declarations of public variables (if used).
You will have to refer to the PAWN booklet "The Language" for writing the declarations mentioned in the above list.

The rationale behind having these declarations in an implicitly included file is that the definitions are now always available. This avoids errors, especially in the case of overloaded operators and public functions. If the definition of an overloaded operator is missing, in many cases the PAWN compiler will use the

default operator without warning. If a user makes a mistake in the declaration of a public function, the host application will not be able to call it, or it will pass the wrong parameters. A forward declaration of the public function catches this error, because the incorrect public function will not match the earlier declaration.

Apart from this implicit include file, the user can also write custom include files and explicitly include these. In addition, a host application may supply additional "system" include files that are not added to a project automatically and must be included explicitly.

The next two chapters are on the PAWN compiler and the abstract machine respectively. The most common set-up is the one where the compiler runs as a separate process that is spawned from the host application.

# The compiler

The PAWN compiler is currently the only translator (or parser) that implements the PAWN language. The PAWN compiler translates a text file with source code to a binary file for an abstract machine. The output file format is in appendix D. The usage of the PAWN compiler is described in the PAWN booklet "The Language".

• Deployment / installation

In most operating systems, the compiler is a separate self-contained executable program. It can run as is, but it will look for a configuration file in the same directory as where the compiler is in itself, and it will locate (system) include files in a specific directory. For the retail packages, the actual compiler is in a dynamically loaded library. This library is called "`libpawnc.so`" or "`libpawnc.dll`" (for Linux/UNIX and "`pawncc.exe`" for Microsoft Windows respectively).

Concretely, to set up the PAWN compiler on a system:
◇ Copy the program file for the compiler (typically "`pawncc`" for Linux/UNIX and "`pawncc.exe`" for Microsoft Windows) in a directory of your choice, and also copy the library "`libpawnc.so`" or "`libpawnc.dll`" if it exists.
◇ Optionally copy or create a configuration file, called "`pawn.cfg`", in the same directory.
◇ Add a subdirectory called "`include`" and copy the include files into that directory —especially add the "DEFAULT.INC" prefix file into that directory, if applicable.* This "include" directory may either be below the directory in which the compiler and `pawn.cfg` reside, or it may be at the same level as the directory where the compile and `pawn.cfg` are. For example, if on a Windows system `pawncc.exe` and `libpawnc` are in `C:\Pawn\bin`, then the compiler will look for include files in either the directory `C:\Pawn\bin\include` or `C:\Pawn\include`.

Prefix file: 1

• The configuration file

On platforms that support it (currently Microsoft DOS, Microsoft Windows and Linux), the compiler reads the options in a "configuration file" on start-up. The

---

* For details on the prefix file, look up the compiler command line option `-p` in the PAWN booklet "The Language".

configuration file must have the name "`pawn.cfg`" and it must reside in the same directory as the compiler executable program and/or the compiler dynamically loaded library.

In a sense, the configuration file is an implicit response file (see the PAWN booklet "The Language" for details on response files). Options specified on the command line may overrule those in the configuration file.

## Errors

### • Compiler errors

The error and warning messages produced by the compiler are described in the companion PAWN booklet "The Language".

### • Run time errors

The function library that forms the abstract machine returns error codes. These error codes encompass both errors for loading and initializing a binary file and run-time errors due to programmer errors (bounds-checking).

# The abstract machine

The abstract machine is a C function library. There are several versions: one that is written in ANSI C, and optimized versions that use GNU C extensions or assembler subroutines.

### • Deployment / installation

The abstract machine is either linked into the host program, or it is implemented as a loadable library (a DLL in Microsoft Windows, or a "shared library" in Linux). No special considerations are required for redistributing the abstract machine.

If you allow extension modules to be loaded dynamically, you may need to set an environment variable in Linux/UNIX. These operating systems search for libraries in a specific path, unless an explicit path is given for the library. In Linux/ UNIX, the abstract machine builds a specific path from the combination of the environment variable "AMXLIB" and the library name. For example, if AMXLIB is set to "/opt/Pawn/bin" and the module uses amxTime, the abstract machine will load "/opt/Pawn/bin/amxTime.so". The name of the environment variable is configurable, —see .

## Using the abstract machine

To use the abstract machine:
1   initialize the abstract machine and load the compiled pseudo-code;
2   register all native functions that the host program provides, directly with amx_Register or indirectly;
3   run the compiled script with amx_Exec;
4   and clean up the abstract machine and other resources.

The example (in C) below illustrates these steps:

```
int main(int argc, char *argv[])
{
  extern AMX_NATIVE_INFO console_Natives[];
  extern AMX_NATIVE_INFO core_Natives[];

  AMX amx;
  cell ret = 0;
  int err;

  if (argc != 2)
    PrintUsage(argv[0]);
```

```
    err = aux_LoadProgram(&amx, argv[1], NULL);
    if (err != AMX_ERR_NONE)
      ErrorExit(&amx, err);

    amx_Register(&amx, console_Natives, -1);
    err = amx_Register(&amx, core_Natives, -1);
    if (err)
      ErrorExit(&amx, err);

    err = amx_Exec(&amx, &ret, AMX_EXEC_MAIN);
    if (err)
      ErrorExit(&amx, err);
    printf("%s returns %ld\n", argv[1], (long)ret);

    aux_FreeProgram(&amx);
    return 0;
  }
```

The `cell` data type is defined in `AMX.H`, it usually is a 32-bit integer.

The program checks first whether a command line argument is present; if so, the program assumes that it is the filename of a compiled PAWN script. The function `PrintUsage` is discussed later in this chapter.

Function `aux_LoadProgram` allocates memory for the abstract machine, loads the compiled pseudo-code and initializes the lot. This function is not part of the PAWN core, just because of what it does: memory allocation and file I/O. Therefore, the function `aux_LoadProgram` is implemented in a separate source file and prefixed with "`aux_`", rather than "`amx_`" ("aux" stands for auxiliary). We will look at an implementation of `aux_LoadProgram` below.

The program has declarations for two sets of native functions: console functions from `AMXCONS.C` and core functions from `AMXCORE.C`. Both these sets are registered with the abstract machine. Function amx_Register returns an error code if the compiled script contains unresolved calls to native functions. Hence, only the result of the *last* call to amx_Register needs to be checked.

The call to amx_Exec runs the compiled script and returns both an error code and a program result code. Errors that can occur during amx_Exec are division by zero, stack/heap collision and other common run-time errors, but a native function or an `assert` instruction in the source code of the PAWN program may also abort the PAWN script with an error code.

Once the script has finished running, `aux_FreeProgram` releases memory and resources that were allocated for it. This, too, is an auxiliary function —see page 10 for an example implementation.

The abstract machine API has no functions that read a compiled script from file into memory; the host program must implement these. An example implementation that comes with the PAWN toolkit is `aux_LoadProgram`. This is a fairly large function as it:

1 opens the file and checks/massages the header;

2 optionally allocates a memory block to hold the compiled pseudo-code (P-code);

3 reads in the complete P-code file;

4 initializes the abstract machine and prepares the P-code for execution;

5 cleans up resources that it allocated in case an error occurs.

```
int aux_LoadProgram(AMX *amx, char *filename, void *memblock)
{
  FILE *fp;
  AMX_HEADER hdr;
  int result, didalloc;

  /* step 1: open the file, read and check the header */
  if ((fp = fopen(filename, "rb")) == NULL)
    return AMX_ERR_NOTFOUND;
  fread(&hdr, sizeof hdr, 1, fp);
  amx_Align16(&hdr.magic);
  amx_Align32((uint32_t *)&hdr.size);
  amx_Align32((uint32_t *)&hdr.stp);
  if (hdr.magic != AMX_MAGIC) {
    fclose(fp);
    return AMX_ERR_FORMAT;
  } /* if */

  /* step 2: allocate the memblock if it is NULL */
  didalloc = 0;
  if (memblock == NULL) {
    if ((memblock = malloc(hdr.stp)) == NULL) {
      fclose(fp);
      return AMX_ERR_MEMORY;
    } /* if */
    didalloc = 1;
    /* after amx_Init(), amx->base points to the memory block */
  } /* if */

  /* step 3: read in the file */
  rewind(fp);
  fread(memblock, 1, (size_t)hdr.size, fp);
  fclose(fp);

  /* step 4: initialize the abstract machine */
  memset(amx, 0, sizeof *amx);
  result = amx_Init(amx, memblock);
```

```
    /* step 5: free the memory block on error, if it was allocated here */
    if (result != AMX_ERR_NONE && didalloc) {
      free(memblock);
      amx->base = NULL;                    /* avoid a double free */
    } /* if */

    return result;
  }
```

**Step 1**: PAWN can run on both Little-Endian and Big-Endian architectures, but it uses a single file format for its pseudo-code. The multi-byte fields in the header of the file format are in Little Endian (or "Intel" format). When running on a Big Endian CPU, function `amx_Init` adjusts all fields in the `AMX_HEADER` structure from Little Endian to Big Endian. The function `aux_LoadProgram`, however, deals with a few header header fields *before* `amx_Init` has run, so it must perform the proper alignment *explicitly* on a Big Endian CPU, using the functions `amx_Align16` and `amx_Align32`. Calling these functions on a Little Endian machine does no harm.

The header of the compiled script contains a special number. We check this "magic file" here immediately, because if we find a different value, all other fields in the header will likely be mangled as well.

**Step 2**: The size of the binary image of the compiled script is not equal to the total memory requirements —it lacks the memory requirements for the stack and the heap. The "`stp`" (Stack Top) field in the header of the file format gives the correct memory size.

With the above implementation of `aux_LoadProgram`, you can load the compiled script either into a block of memory that you allocated earlier, or you can let `aux_LoadProgram` allocate memory for you. The `memblock` argument must either point to a memory block with an adequate size, or it must be `NULL`, in which case the function allocates a block.

**Step 3**: The complete file must be read into the memory block, including the header that we read near the function. After reading the file into memory, it can be closed. As an aside, the the value of `hdr.size` is the same as the file length.

**Step 4**: It is important to clear the `AMX` structure before calling `amx_Init`, for example using `memset`.

**Step 5**: `amx_Init` does a few checks on the header and it runs quickly through the P-code to relocate jump and variable addresses and to check for invalid in-

structions. If this verification step fails, we will want to free the memory block that the function allocated, but *only* if the function allocated it.

Finally, for completeness, the functions `aux_FreeProgram`, `ErrorExit` and `Print-Usage` are below:

```
int aux_FreeProgram(AMX *amx)
{
  if (amx->base!=NULL) {
    amx_Cleanup(amx);
    free(amx->base);
    memset(amx, 0, sizeof(AMX));
  } /* if */
  return AMX_ERR_NONE;
}

void ErrorExit(AMX *amx, int errorcode)
{
  printf("Run time error %d: \"%s\" on address %ld\n",
         errorcode, aux_StrError(errorcode),
         (amx != NULL) ? amx->cip : 0);
  exit(1);
}

void PrintUsage(char *program)
{
  printf("Usage: %s <filename>\n", program);
  exit(1);
}
```

## • Controlling program execution

The code snippets presented above are enough to form an interpreter for PAWN programs. A drawback, however, is that the PAWN program runs uncontrolled once it is launched with amx_Exec. If the PAWN program enters an infinite loop, for example, the only way to break out of it is to kill the complete interpreter —or at least the thread that the interpreter runs in. Especially during development, it is convenient to be able to abort a PAWN program that is running awry.

The abstract machine has a mechanism to monitor the execution of the pseudo-code that goes under the name of a "debug hook". The abstract machine calls the debug hook, a function that the host application provides, at specific events, such as the creation and destruction of variables and executing a new statement. Obviously, the debug hook has an impact on the execution speed of the abstract machine. To minimize the performance loss, the host application can enable the debug hook "as needed" and keep it disabled when it is not needed.

To install a debug hook, call `amx_SetDebugHook`. A debug hook function can inspect the status of the abstract machine and browse through the symbolic information (and the source files) when it gets invoked. To set up a debug hook, you would add a call to `amx_SetDebugHook` somewhere between `amx_Init` and `amx_Exec`. In the PAWNRUN program laid out at page 6 (function `main`), you could add the following line below the call to `aux_LoadProgram`:

```
err = amx_SetDebugHook(&amx, aux_Monitor);
```

The function `amx_Monitor` becomes the "debug hook" function that is attached to the specified abstract machine. A minimal implementation of this function is below:

```
int AMXAPI aux_Monitor(AMX *amx)
{
  return abortflagged ? AMX_ERR_EXIT : AMX_ERR_NONE;
}
```

If the debug hook returns any code other than `AMX_ERR_NONE`, execution halts and `amx_Exec` returns the specific error code. The code `AMX_ERR_SLEEP` is a special case: it aborts execution in a way that it can be *restarted* by passing the special "index" `AMX_EXEC_CONT` to function `amx_Exec`. The abstract machine calls the debug hook just before executing a new statement (on a new line).

Exactly *how* the host program decides whether to continue running or to abort the abstract machine is implementation dependent. This example uses a global variable, `abortflagged`, that is set to a non-zero value —by some magical procedure— if the abstract machine(s) must be aborted.

There exists a more or less portable way to achieve the "magic" referred to in the previous paragraph. If you set up a `signal` function to set the `abortflagged` variable to 1 on a `SIGINT` signal, you have an "ANSI C"-approved way to abort an abstract machine. The snippet for the signal function appears below:

```
void sigabort(int sig)
{
  abortflagged = 1;
  signal(sig, sigabort);    /* re-install the signal handler */
}
```

And somewhere, before calling `amx_Exec`, you add the line:

```
signal(SIGINT, sigabort);
```

Debug hook functions allow you to monitor stack usage, profile execution speed at the source line level and, well... write a debugger. Detailed information on the debug hook is found in appendix E of this manual.

One caveat is that the debug hook depends on the presence of BREAK opcodes. When a PAWN program is compiled *without* debug information, no BREAK opcodes are present in the P-code and the debug hook will never get called. That renders our monitor function ineffective. This is okay, though, because the user or the host application has explicitly compiled *without* debugging checks to improve run-time performance —by default the PAWN compiler has (minimal) debug information enabled.

In your host application, you may want to check for debug information and, in its absence, warn the user that some functionality may not be available. To verify whether debug information is present, use the following snippet:

```
uint16_t flags;
amx_Flags(&amx, &flags);
if ((flags & AMX_FLAG_NOCHECKS) != 0) {
  /* no BREAK opcodes are present, a debug hook will not run */

    Code to handle the case of missing debug support

} /* if */
```

## • A smarter "break-out" hook

The debug hook described above has a major drawback: it makes the script run more slowly, because the abstract machine calls the hook function repetitively. In the normal situation (no "abort" is signalled), the hook function does nothing —except take time.

An improvement is to run *without* debug hook normally, and to set up a debug hook only *after* an abort signal break has already been detected. To this end, we change the `signal` function to:

```
void sigabort(int sig)
{
  /* install the debug hook procedure if this was not done already */
  amx_SetDebugHook(global_amx, aux_Monitor);
  signal(sig, sigabort);        /* re-install the signal handler */
}
```

If you use the debug hook only to check for a "break" or "abort" signal, there is
no need for a global variable that flags this request: de debug hook will only ever
be called when the user has already issued the break/abort request, so the debug
hook can just always return with an error code to cause the script to abort.

```
int AMXAPI aux_Monitor(AMX *amx)
{
  return AMX_ERR_EXIT;
}
```

While the `abortflagged` global variable is was made redundant, I have introduced
a new global variable: `global_amx`. The standard `signal` function from ANSI
C does not provide for passing a "user value" via a parameter, so you have to
invent another way to make the abstract machine that you wish to abort known
to the `signal` function. In practice, your host application will likely have another
implementation for the `signal` function, such as an event procedure in a GUI.

## • Monitoring stack/heap usage

A useful function that the debug hook can implement is to monitor how much
memory the compiled script uses at run-time —in other words, checking the max-
imum stack and heap usage. To this end, the example below extends the debug
"monitor" function of the previous sections, and adds another refinement at the
same time.

```
int AMXAPI amx_Monitor(AMX *amx)
{
  int err;
  unsigned short flags;
  STACKINFO *stackinfo;

  /* record the heap and stack usage */
  err = amx_GetUserData(amx, AMX_USERTAG('S','t','c','k'),
                        (void**)&stackinfo);
  if (err == AMX_ERR_NONE) {
    if (amx->stp - amx->stk > stackinfo->maxstack)
      stackinfo->maxstack = amx->stp - amx->stk;
    if (amx->hea - amx->hlw > stackinfo->maxheap)
      stackinfo->maxstack = amx->stp - amx->stk;
  } /* if */

  /* check whether an "abort" was requested */
  return abortflagged ? AMX_ERR_EXIT : AMX_ERR_NONE;
}
```

This extended version of `amx_Monitor` still checks the `abortflagged` variable (which is set on a Ctrl-C or Ctrl-Break signal), but at the same time it also calculates the current stack and heap usage and records these in a structure. The used stack space is the difference between the top-of-stack and the current stack point; similarly, the heap usage is the difference between the current heap pointer and the heap bottom. More interesting is that the function stores this maxima of the calculated values in the variable `stackinfo`, which is a structure with the following definition:

```
typedef struct tagSTACKINFO {
  long maxstack, maxheap;
} STACKINFO;
```

The abstract machine allows a host application to set one or more "user values". In the current implementation of the abstract machine, up to four user values may be used. To indicate which of the user values you want to access, it is convenient to use the macro `AMX_USERTAG` with a four-letter identification string. In this example, the identification characters are 'S','t','c','k'.

The monitor function only retrieves a pointer to the `stackinfo` structure and updates its fields. Elsewhere in the program, before the call to amx_Exec, the following lines are present to initialize the variable and set its address as a user value:

```
STACKINFO stackinfo;
memset(&stackinfo, 0, sizeof stackinfo);
err = amx_SetUserData(&amx, AMX_USERTAG('S','t','c','k'), &stackinfo);
```

As you will probably want to monitor the stack usage from the start, the debug hook has also to be set up *before* calling amx_Exec. Setting up the debug hook is covered on page 10.

## • Preparing for memory-hungry scripts

The core run-time files that build the abstract machine executor (`AMX.C` and `AMXEXEC.ASM`) are specifically designed *not* to use dynamic memory or to rely on a particular memory allocator.* The reasoning behind this design is that the abstract machine executor is made to be linked into host applications and, in practice, diverse host applications use dissimilar memory allocation schemes —from instrumented versions of `malloc` to garbage collection algorithms.

---

\* There are a few "violations" of this design: the "property" functions in AMXCORE.C call "malloc"; that said, native functions are considered *non-core* functions.

The drawback of this design, however, is that the address range that a compiled script runs in cannot easily grow: the executor itself cannot grow the memory block because it knows nothing about the memory allocator that the host program uses, and the host program will have to reach into the internals of the abstract machine executor after it resizes the memory block. Already determining *when* to grow the block is involved. Hence, the address range that a script can use should be seen as "fixed" or static.

The problem is that the host application cannot foresee what kind of scripts users will write and how much breathing room their scripts need. A user may set this value him/herself with #pragma dynamic, but this involves guesswork and it is not user friendly. When the host program also runs the compiler, it can set the heap/stack size to a value that is large enough for every imaginable script, but at the risk that expanding the memory footprint of the host program by this size impacts the general performance of the complete system (read "causes excessive swapping").

For "#pragma dynamic" and compiler options: see the Pawn booklet "The Language"

Modern operating systems allow for an efficient solution for this dilemma: allocate the memory address range without reserving the memory and subsequently reserve (or "commit") the memory on an as-needed basis. The code snippets in this section are for the "Win32" family of Microsoft Windows, but the concept applies to many operating systems that provide virtual memory.

```
int main(int argc,char *argv[])
{
  size_t memsize;
  void *program;
  AMX amx;
  cell ret = 0;
  int err;

  if (argc != 2 || (memsize = aux_ProgramSize(argv[1])) == 0)
    PrintUsage(argv[0]);

  program = VirtualAlloc(NULL, memsize, MEM_RESERVE, PAGE_READWRITE);
  if (program == NULL)
    ErrorExit(NULL, AMX_ERR_MEMORY);

  __try {

    err = aux_LoadProgram(&amx, argv[1], program);
    if (err)
      ErrorExit(&amx, err);

    amx_ConsoleInit(amx);
    err = amx_CoreInit(amx);
    if (err)
      ErrorExit(&amx, err);
```

```
    err = amx_Exec(&amx, &ret, AMX_EXEC_MAIN);
    if (err)
      ErrorExit(&amx, err);
    if (ret != 0)
      printf("%s returns %ld\n", argv[1], (long)ret);

  } __except (aux_CommitMemory(GetExceptionInformation(), program, memsize)){
    /* nothing */
  } /* try */

  amx_ConsoleCleanup(&amx);
  amx_CoreCleanup(&amx);
  amx_Cleanup(&amx);
  VirtualFree(program, memsize, MEM_DECOMMIT);
  VirtualFree(program, 0, MEM_RELEASE);
  return 0;
}
```

The above `main` function is a variation of the one on . Instead of using `malloc` and `free` (indirectly through `aux_LoadProgram` and `aux_FreeProgram`), it calls the Win32 functions `VirtualAlloc` and `VirtualFree`. The call to `VirtualAlloc` reserves an address range, but does not "commit" the memory, meaning that no memory is allocated at this point. Later, one may commit chunks of memory inside this address range, with the advantage that one can now specify the memory address that must be committed. At the end of the program, `VirtualFree` must be called twice, as the function can only release memory in one call if it has either been fully committed or fully decommitted. The first call to `VirtualFree` decommits all committed memory.

When a program tries to access memory that is not committed, an "access violation" exception occurs. Function `main` catches exceptions and handles them in the function below. Note that the function carefully checks whether it gets an exception that it can handle. PAWN typically accesses elements in cells, so that is the default size to commit (variable `elemsize` in the code snippet below), but this size is adjusted if it would exceed the allocate memory range.

```
DWORD aux_CommitMemory(struct _EXCEPTION_POINTERS *ep, void *memaddr,
                       size_t memsize)
{
  void *virtaddr;
  int elemsize;

  if (ep->ExceptionRecord->ExceptionCode != EXCEPTION_ACCESS_VIOLATION)
    return EXCEPTION_CONTINUE_SEARCH;

  virtaddr = (void*)ep->ExceptionRecord->ExceptionInformation[1];
  if (virtaddr < memaddr || virtaddr >= ((char*)memaddr + memsize))
    return EXCEPTION_CONTINUE_SEARCH;
```

```
    elemsize = sizeof(cell);
    if ((char*)virtaddr + elemsize > (char*)memaddr + memsize)
      elemsize = ((char*)memaddr + memsize) - (char*)virtaddr;

    if (VirtualAlloc(virtaddr, elemsize, MEM_COMMIT, PAGE_READWRITE) == NULL)
      return EXCEPTION_CONTINUE_SEARCH;

    return EXCEPTION_CONTINUE_EXECUTION;
  }
```

With these modifications, a host program (or a user) can now specify a size for the stack and heap of a few megabytes when compiling a script file, and be assured that *only* the memory that the program *really* uses is ever allocated. Microsoft Windows commits memory blocks in "pages", which are 4 kiB in size. That is, although the above code commits only one `cell` (4 bytes), a range of 1024 cells get committed.

A host program may choose to periodically decommit all memory for a running script, in order to reduce the memory footprint of the script (this is not implemented in the above code snippet).

Another change in `main` in comparison with the first implementation at is that it calls the functions `amx_ConsoleInit` and `amx_CoreInit` rather than `amx_Register` directly. As is explained in the section on writing extension modules (an *extension module* is a native function library), it is proposed that an extension module provides initialization and clean-up functions; the initialization function registers the native functions.

## Calling "public" functions

The implementations presented so far would only call the function `main` in a compiled PAWN script. Many implementations require multiple entry points and need to be able to pass input parameters to that entry point. We need two steps to enable this:
◇ The script must provide one or more public functions.
◇ The host program must be adapted to locate the public function and pass its index (and parameters) to `amx_Exec`.

To start with the latter step, the host program is adapted so that it finds a particular public function by name. Function `amx_Exec` takes an index of a public function as a parameter; the previous examples used the special constant `AMX_EXEC_MAIN` to start with the "main" entry point. If you know the name of the public function,

`amx_FindPublic` returns its index. For this purpose, include the snippet below before the call to `amx_Exec` (it assumes that the name of the public function is in the variable `argv[2]`):

```
err = amx_FindPublic(&amx, argv[2], &index);
if (err)
  ErrorExit(&amx, err);
```

A public function may require input arguments. If so, these must be "pushed" onto the AMX stack prior to calling `amx_Exec`. For a numeric parameter that is passed by value, the sequence would be:

```
cell value = 123;

amx_Push(&amx, value);
amx_Exec(&amx, NULL, index);
```

Numeric ("passed-by-value") parameters are removed automatically from the stack when `amx_Exec` returns. When the parameter is a reference parameter or an array (or a string), the memory needs to be explicitly freed after `amx_Exec` returns —this gives the host application the ability to inspect the value(s) that the script stored in these parameters. For example, to pass a string from `argv[3]` in the host program to a public function in the abstract machine, use a snippet like the following:

```
cell *address;

/* pass argv[3] as input */
amx_PushString(&amx, &address, argv[3], 0, 0);
amx_Exec(&amx, NULL, index);

/* copy the (possibly changed) string out of the AMX (optional) */
char resultstring[128];
amx_GetString(resultstring, address, 0, sizeof(resultstring));

/* release the memory */
amx_Release(&amx, address);
```

The above snippet passes the string as an "unpacked" string, meaning that in the script, every cell holds one character. The PAWN API (and the PAWN language itself) support "wide characters" for Unicode applications. The above example assumes a non-Unicode environment —in an Unicode environment the next-to-last parameter to `amx_SetString` and `amx_GetString` should be non-zero.

In addition to storing the input argument on the stack, function `amx_PushString` returns the address of the memory block that it allocated in the abstract machine. This address must be passed to `amx_Release` to free the allocated memory. The host program can also use this address to inspect or copy data out of the

abstract machine after `amx_Exec` returns. In this example, the host program calls `amx_GetString` to store the string that the script modified into a local variable.

If a public function has a variable argument list, all parameters in this list must be passed by reference. That is, you have to follow the above procedure for any argument that falls in the variable argument list of the public function. For reference arguments, pass an array with a size of one cell.

Below is the complete `main` function of a run-time that allows you to execute any public function and pass in a string. This program is, again, a modification of the example program on page 6. It includes the calls to `amx_FindPublic` and `amx_PushString` mentioned above, and it also shows how to pass one extra parameter through `amx_Exec`.

```c
int main(int argc, char *argv[])
{
  size_t memsize;
  void *program;
  AMX amx;
  int index, err;
  cell *address;
  char output[128];

  if (argc != 4 || (memsize = aux_ProgramSize(argv[1])) == 0)
    PrintUsage(argv[0]);

  program = malloc(memsize);
  if (program == NULL)
    ErrorExit(NULL, AMX_ERR_MEMORY);

  err = aux_LoadProgram(&amx, argv[1], program);
  if (err)
    ErrorExit(&amx, err);

  amx_ConsoleInit(&amx);
  err = amx_CoreInit(&amx);
  if (err)
    ErrorExit(&amx, err);

  err = amx_FindPublic(&amx, argv[2], &index);
  if (err)
    ErrorExit(&amx, err);

  err = amx_PushString(&amx, &address, argv[3], 0, 0);
  if (err)
    ErrorExit(&amx, err);

  err = amx_Exec(&amx, NULL, index);
  if (err)
    ErrorExit(&amx, err);

  amx_GetString(output, address, 0, sizeof output);
  amx_Release(&amx, address);
  printf("%s returns \"%s\"\n", argv[1], output);
```

```
    amx_ConsoleCleanup(&amx);
    amx_CoreCleanup(&amx);
    amx_Cleanup(&amx);
    free(program);
    return 0;
}
```

When the program returns from `amx_Exec`, the host program can inspect the returned value(s) and free the allocated space. The program presented here uses `amx_GetString` to retrieve the string that the public function (possibly) modified. The function `amx_Release` frees the memory allocated by `amx_PushString`. When you pass in multiple string or array arguments to a public function, a single call to `amx_Release` can free all allocated memory, see the function description at page 41.

To demonstrate this program, we must also write a script that contains a public function and that accepts a string parameter. Below is a variation of the "ROT13" example script from the PAWN booklet "The Language". The essential modification is the keyword `public` that is prefixed to the function name "`rot13`" —and the removal of the `main` function which has now become redundant.

```
public rot13(string[])
    {
    for (new index = 0; string[index]; index++)
        if ('a' <= string[index] <= 'z')
            string[index] = (string[index] - 'a' + 13) % 26 + 'a'
        else if ('A' <= string[index] <= 'Z')
            string[index] = (string[index] - 'A' + 13) % 26 + 'A'
    }
```

With these modifications, and supposing that we have built the C program to an executable with the name "pawnrun", we can execute the script with:

```
pawnrun rot13.amx rot13 hello-world
```

Essentially the same procedure as outlined above applies to the passing of non-string arrays to a public function:

1 pass the array to the abstract machine with `amx_PushArray`;

2 call the public function;

3 optionally copy the array back, out of the abstract machine —using the pointer that `amx_PushArray` returned;

4 free the memory block in the abstract machine with `amx_Release`, again passing it the pointer that `amx_PushArray` returned.

The implementation of "pawnrun" that calls the ROT13 script (page 19) uses the functions `amx_SetString` and `amx_GetString` to copy strings into and out of the abstract machine. The reasons for using these functions has to do with the difference in memory layout of strings in C/C++ versus PAWN. When passing arrays of integers (`cell`-sized) or floating point values, you can just use the standard C functions `memmove` and `memcpy`.

For an example, imagine a host application that does some statistical processing of lists of floating point numbers, and that allows users of the application to "customize" the operation by providing an alternative implementation of key routines in a PAWN script. In particular, the host application allows user to override the "mean" calculation with a script that contains the public function `CalculateMean` with the following signature:

```
public Float: CalculateMean(Float: values[], items)
```

This is what the host application does (I am showing only a snippet of code here, rather than a complete implementation of a C/C++ function; refer to page 19 for the context of this snippet):

```
float Mean;        /* the variable holding the result of the calculation */
float Values[];    /* array with the numbers to get the mean of */
int Number;        /* number of elements in "Values" */

AMX amx;           /* the abstract machine, already initialized */
int index, err;
cell *address;

err = amx_FindPublic(&amx, "CalculateMean", &index);
if (err != AMX_ERR_NONE) {
  /* custom function not present, use a built-in function to
   * calculate the mean
   */
  Mean = CalculateStdMean(Values, Number);
} else {

  /* 1. push the second argument to the public function first (arguments
   *    must be pushed in reverse order)
   */
  amx_Push(&amx, Number);

  /* 2. allocate memory in the abstract machine */
  err = amx_PushArray(&amx, &address, Values, Number);
  if (err == AMX_ERR_NONE) {

    /* 3. call the public function with the "AMX address" */
    err = amx_Exec(&amx, (cell*)&Mean, index);
    if (err != AMX_ERR_NONE)
      printf("Run time error %d on line %ld\n", err, amx.curline);

    /* 4. we could copy the array back here, but it is not very
     *    useful in this particular case */
```

```
    /* 5. release memory in the abstract machine */
    amx_Release(&amx, address);

  } else {
    printf("Failed to allocate %d cells\n", Number);
    Mean = 0.0;
  } /* if */

} /* if */
```

This example may appear to serve a bizarre purpose: "Why have the user customize the *mean* function? What kind of alternative *mean* function can a user invent that is not absurd or fraudulent?" —until you dive into the subject and discover a full and complex world behind a simple concept as "the mean". The most well known and most frequently used kind of average, which has become synonymous with *the mean*, is the "arithmetic average":* the sum of all elements divided by the number of elements. It is well known that the arithmetic average is sensitive to outliers, e.g. coming from noisy data, and in such cases the "median" is often proposed as a stable alternative to the (arithmetic) mean.

The median and the mean are the two extremities of the (arithmetic) "trimmed mean". The trimmed mean throws out the lowest and the highest few samples and calculates the arithmetic average over the remainder. The number of discarded samples is a parameter of the *trimmed mean* function: if you discard zero samples what you get is the standard mean and if you discard all but one sample, the remaining sample is the median.

The example implementation of a trimmed mean below discards only the top and bottom samples. This particular configuration of the trimmed mean has become known as the "Olympic mean", referring to a similar procedure that has in the past been used to establish the average performance of athletes.

```
    #include <float>

    public Float: CalculateMean(Float: values[], items)
        {
        /* return a "trimmed mean" by throwing out the minimum and
         * the maximum value and calculating the mean over the remaining
         * items
         */
        assert items >= 3    /* should receive at least three elements */
```

---

* Other kinds are the geometric average, the harmonic average and the "root mean square".

```
new Float: minimum = values[0]
new Float: maximum = values[0]
new Float: sum = 0.0
for (new i = 0; i < items; i++)
    {
    if (minimum > values[i])
        minimum = values[i]
    else if (maximum < values[i])
        maximum = values[i]
    sum += values[i]
    }
return (sum - minimum - maximum) / (items - 2)
}
```

This concludes handling array and string arguments to a public function by the host application; what is left are *reference arguments*. This does not need an in-depth discussion, however, because the host application can handle a reference argument as an array argument with the size of one (1) `cell`.

## Function reference

With one exception, all functions return an error code if the function fails (the
exception is `amx_NativeInfo`). A return code of zero means "no error".

---

amx_Address                                    Expand a truncated address

Syntax:      `[cell*] amx_Address([AMX*] amx, [cell] param)`

amx          The abstract machine.

param        A field from the `params` argument of a native function.

Returns:     The `param` argument cast to a "pointer-to-cell", after possibly ex-
             panding the argument from a truncated pointer (see notes).

Notes:       When the size of a cell is big enough to hold a pointer (such as in
             a 32-bit program that uses 32-bit cells for PAWN), this **macro** does
             merely a cast of argument `param` from "`cell`" to "`cell*`".

             When a pointer does not fit in a cell, however, this **macro** expands
             the truncated address in `param` to a full address using the high bits
             of the data pointer in the AMX structure. You may need to use this
             function, for example, when running a 32-bit PAWN configuration in
             a 64-bit host application).

             Using the macro `amx_Address` therefore increases portability, but it
             is not needed in the typical case that the pointer size and cell size
             are equal.

---

amx_Align16/32/64  Conditionally swap bytes in a 16-bit, 32-bit or 64-bit word

Syntax:      `uint16_t *amx_Align16(uint16_t *v)`

             `uint32_t *amx_Align32(uint32_t *v)`

             `uint64_t *amx_Align64(uint64_t *v)`

v            A pointer to the 16-bit value, the 32-bit value or the
             64-bit value whose bytes must be aligned.

Notes:        Multi-byte fields in the header in the compiled file are in Little
              Endian format.  If run on a Big Endian architecture, these two
              functions function swap the bytes in a 16-bit/32-bit/64-bit Little
              Endian word. The value v remains unchanged if the code runs on
              a Little Endian CPU, so there is no harm in always calling this
              function.

              The `amx_Align64` is not available in all configurations. If the PAWN
              Abstract Machine was built with for a 16-bit architecture, it is likely
              absent.

See also:     amx_AlignCell

---

## amx_AlignCell                                    Conditionally swap bytes in a cell

Syntax:       `[cell] *amx_AlignCell([cell] *v)`

              v             A pointer to the "cell" value whose bytes must be
                            aligned.

Notes:        This **macro** maps to function `amx_Align16` when a cell is 16-bit,
              to function `amx_Align32` when a cell is 32-bit, and to function
              `amx_Align64` when a cell is 64-bit.

See also:     amx_Align

---

## amx_Allot                                    Reserve heap space in the abstract machine

Syntax:       `int amx_Allot(AMX *amx, int cells, cell **address)`

              amx           The abstract machine.

              cells         The number of cells to reserve.

              address       The address of the cell for C/C++ programs to access.

Notes:   `amx_Allot` allocates memory cells inside the abstract machine and it returns the address relative to the host program's address space. So a C/C$^{++}$ program can use this address and write into the allocated memory. A PAWN function can only access memory inside its abstract machine. If a parameter is to be passed "by reference" to a PAWN function, one must pass the address of that parameter to amx_Exec. In addition, that address *itself* must be within the address range of the abstract machine too.

When passing the address of the allocated memory block to a PAWN function, you should call amx_PushAddress. This function relocates the passed-in address to be relative to the data section of the PAWN program.

After amx_Exec returns, you may inspect the memory block (the PAWN function called by amx_Exec may have written into it) and finally release it by calling amx_Release.

This function is a low-level interface. For passing arrays, strings or reference parameters to a PAWN function, it is usually more convenient to use the functions amx_PushArray and amx_PushString.

See also:   amx_Exec, amx_PushAddress, amx_PushArray, amx_PushString, amx_Release

---

## amx_Callback                                    The default callback

Syntax:   ```
int amx_Callback(AMX *amx, cell index, cell *result,
                 const cell *params)
```

amx       The abstract machine.

index     Index into the native function table; it points to the requested native function.

result    The function result (of the native function) should be returned through this parameter.

params    The parameters for the native function, passed as a list of long integers. The first number of the list is the number of bytes passed to the native functions (from which the number of arguments can be computed).

Returns: The callback should return an error code, or zero for no error. When the callback returns a non-zero code, `amx_Exec` aborts execution.

Notes: The abstract machine has a default callback function, which works in combination with `amx_Register`. You can override the default operation by setting a different callback function using function `amx_SetCallback`.

If you override the default callback function, you may also need to provide an alternative function for `amx_Registers`.

See also: `amx_Exec`, `amx_RaiseError`, `amx_SetCallback`

---

## amx_ctof                                                        Cast "cell" to "float"

Syntax: `[float] amx_ctof([cell] c)`

    `c`              The value to cast from "cell" type to "float".

Returns: The same bit pattern, but now as a floating point type.

Notes: This **macro** casts a "cell" type into a "float" type *without* changing the bit pattern. A normal type cast in C/C++ changes the memory representation of the expression so that its numeric value in IEEE 754 format comes closest to the original integer value. The PAWN parser and abstract machine store floating point values in a cell — when retrieving a floating point value from a cell, the bit pattern must *not* be changed.

See also: `amx_ftoc`

---

## amx_Exec                                                                  Run code

Syntax: `int amx_Exec(AMX *amx, long *retval, int index)`

    `amx`         The abstract machine from which to call a function.

    `retval`     Will hold the return value of the called function upon return. This parameter may be `NULL` if you are not interested in the return value.

| | |
|---|---|
| index | An index into the "public function table"; it indicates the function to execute. See `amx_FindPublic` for more information. Use `AMX_EXEC_MAIN` to start executing at the `main` function, and `AMX_EXEC_CONT` to continue execution from a "sleep state". |

Notes: This function runs the script, starting at the indicated function. It calls the callback function for any native function call that the code in the AMX makes. `amx_Exec` assumes that all native functions are correctly initialized with `amx_Register`.

See also: `amx_FindPublic`, `amx_Register`

---

**amx_FindNative**            Return the index of a native function

Syntax: `int amx_FindNative(AMX *amx, char *name, int *index)`

| | |
|---|---|
| amx | The abstract machine. |
| name | The name of the native function to find. |
| index | Upon return, this parameter holds the index of the requested native function. |

Notes: The returned index is the same as what the abstract machine would pass to `amx_Callback`.

See also: `amx_Callback`, `amx_FindPublic`, `amx_GetNative`, `amx_NumNatives`

---

**amx_FindPublic**            Return the index of a public function

Syntax: `int amx_FindPublic(AMX *amx, char *name, int *index)`

| | |
|---|---|
| amx | The abstract machine. |
| name | The name of the public function to find. |
| index | Upon return, this parameter holds the index of the requested public function. |

See also: `amx_Exec`, `amx_FindNative`, `amx_FindPubVar`, `amx_GetPublic`, `amx_NumPublics`

---

**amx_FindPubVar**          Return the address of a public variable

Syntax:      `int amx_FindPubVar(AMX *amx, char *name, cell **address)`

         `amx`       The abstract machine.

         `name`     The name of the public variable to find.

         `address`   Upon return, this parameter holds the variable address, or `NULL` or error.

See also:    `amx_FindPublic`, `amx_GetPubVar`, `amx_NumPubVars`

---

**amx_Flags**          Return various flags

Syntax:      `int amx_Flags(AMX *amx,unsigned short *flags)`

         `amx`       The abstract machine.

         `flags`     A set of bit flags is stored in this parameter. It is a set of the following flags:

                 `AMX_FLAG_DEBUG`
                     the program contains symbolic information

                 `AMX_FLAG_NOCHECKS`
                     the compiled P-code does not include BREAK opcodes, line number information or run-time (bounds) checks

                 `AMX_FLAG_OVERLAY`
                     the program uses overlays

                 `AMX_FLAG_SLEEP`
                     the program uses the `sleep` instruction, which means that the script may be re-entrant

Notes:      A typical use for this function is to check whether the compiled program contains symbolic (debug) information. There is may not be much use in running a debugger without having symbolic information for the program to debug; if the program does not even have contain BREAK opcodes, installing a debugger callback may be skipped altogether.

---

**amx_ftoc**                                                    Cast "float" to "cell"

---

Syntax:        `[cell] amx_ftoc([float] f)`

        `f`         The value to cast from "float" type to "cell".

Returns:       The same bit pattern, but now as a "cell" type.

Notes:         This **macro** casts a "float" type into a "cell" type *without* changing
               the bit pattern. A normal type cast in C/C$^{++}$ changes the memory
               representation of the expression so that its numeric value in integer
               format is the integral (truncated) value of the original rational value.
               The PAWN parser and abstract machine store floating point values in
               a cell —when storing a floating point value in a cell, the bit pattern
               must *not* be changed.

See also:      amx_ctof

---

**amx_GetNative**                                  Return a native function name

---

Syntax:        `int amx_GetNative(AMX *amx, int index, char *name)`

        `amx`       The abstract machine.

        `index`     The index of the requested function. Use zero to re-
               trieve the name of the first native function.

        `name`      The string that will hold the name of the native func-
               tion.

Notes:         The string should be large enough to hold longest function name
               plus the terminating zero byte. Use amx_NameLength to inquire this
               length.

See also:      amx_FindNative, amx_GetPublic, amx_NameLength,
               amx_NumNatives

## amx_GetPublic                          Return a public function name

Syntax:     `int amx_GetPublic(AMX *amx, int index, char *name, ucell`
            `          *address)`

`amx`        The abstract machine.

`index`      The index of the requested function. Use zero to re-
            trieve the name of the first public function.

`name`       The string that will hold the name of the public func-
            tion.

`address`    This parameter will hold the address of the function
            on return. This parameter may be `NULL` on entry.

Notes:      The string should be large enough to hold longest function name
            plus the terminating zero byte. Use amx_NameLength to inquire this
            length.

            The address returned in parameter `address` is relative the the code
            segment of the abstract machine (it must be relocated before a host
            application can use it). Some extensions, like the overlay system,
            uses the function address to mark particular functions as "non-
            swappable" —see appendix J on page 130.

See also:   amx_FindPublic, amx_GetPubVar, amx_NameLength,
            amx_NumPublics

## amx_GetPubVar                 Return a public variable name and address

Syntax:     `int amx_GetPubVar(AMX *amx, int index, char *name,`
            `          cell **address)`

`amx`        The abstract machine.

`index`      The index of the requested variable. Use zero to re-
            trieve the name and address of the first public variable.

`name`       The string that will hold the name of the public vari-
            able.

|  | address | Upon return, this parameter holds the variable address. |

Notes:      The string should be large enough to hold longest variable name plus the terminating zero byte. Use amx_NameLength to inquire this length.

See also:    amx_FindPubVar, amx_GetPublic, amx_NameLength, amx_NumPubVars

---

**amx_GetString**                 Retrieve a string from the abstract machine

Syntax:
```
int amx_GetString(char *dest, cell *source,
                  int use_wchar, size_t size)
```

| | dest | A pointer to a character array of sufficient size to hold the converted source string. |
| | source | A pointer to the source string. |
| | use_wchar | A non-zero value interprets the dest argument as a pointer to "wide characters" —i.e. wchar_t, regardless of its char type. This allows the function to store Unicode strings. |
| | size | The maximum number of characters to store in dest, including the terminating zero byte. If the string in the source is longer, the string in dest will be truncated. |

Notes:      This function converts both packed strings and unpacked strings from the "PAWN" format to the "C" format. When retrieving an unpacked string with parameter use_wchar set to zero, the function may truncate characters from wide characters to 8-bit ASCII/ANSI.

See also:    amx_SetString

---

**amx_GetUserData**                                      Return general purpose user data

Syntax:        `int amx_GetUserData(AMX *amx, long tag, void **ptr)`

            `amx`         The abstract machine.

            `tag`         The "tag" of the user data.

            `ptr`         Will hold a pointer to the requested user data upon return.

Notes:        The AMX stores multiple "user data" fields. Each field must have a unique tag. The tag may be any value (as long as it is unique), but it is usually formed by a four-letter mnemonic through the macro `AMX_USERTAG`.

            The AMX does not use "user data" in any way. The storage can be used for any purpose.

See also:      amx_SetUserData

---

**amx_Init**                              Create an abstract machine, load the binary file

Syntax:        `int amx_Init(AMX *amx, void *program)`

            `amx`         This variable is initialized with the specific settings of the abstract machine. Before calling this function, all fields of the `amx` structure variable should be set to zero, or be explicitly initialized to relevant values (see the notes).

            `program`  A pointer to the P-code stream of the program. This memory block should remain valid while the abstract machine is in use.

Notes:        `amx_Init` initializes the abstract machine with the settings from the binary file. The binary file must be stored in parameter `program`.

            To have the abstract machine run with the default parameters, set the `amx` structure variable to all zeros before calling this function. When the `program` parameter refers to a block of non-modifiable memory (running from ROM), however, you should initialize the `data` field of the `amx` structure to a separate memory block that resides in RAM.

See also:     amx_Cleanup, amx_InitJIT

---

**amx_InitJIT**                       Compile an abstract machine to native code

Syntax:     int amx_InitJIT(AMX *amx, void *reloc_table,
                            void *native_code)

            amx          The abstract machine, that must already have been
                         initialized with amx_Init.

            reloc_table  A pointer to a block that the JIT compiler can use
                         to create the relocation table. This block is only used
                         during JIT compilation and may be freed as soon as
                         the amx_InitJIT function returns. The size of the
                         block must be at least amx->reloc_size bytes.

            native_code  A pointer to a block that will hold the native code
                         after this function returns. This pointer must be set
                         as the new "base" pointer of the abstract machine (see
                         the notes below).

Notes:      Function amx_Init fills in two fields in the AMX structure that are
            needed for JIT compilation: codesize and reloc_size. Both fields
            are sizes of buffers that must be allocated for amx_InitJIT. The ab-
            stract machine will be compiled into the block native_code, which
            must have the size codesize (or larger) and the JIT compiler needs
            an auxiliary block during compilation, which is reloc_table with
            the size reloc_size.

            The host application is responsible for allocating and freeing the
            required blocks.

            Function amx_Init gives a *conservative minimal estimate* of the
            required code size for the native instructions —meaning that this
            value is (or should be) too large. Function amx_InitJIT adjusts
            the codesize field to the accurate value. After the amx_InitJIT
            function returns, the compiled code needs to be attached to the
            amx structure, and you may want to shrink the memory block to
            the accurate size before doing so. To attach the native code to the
            abstract machine, assign the native_code pointer to the "base"
            field of the amx structure.

On some architectures, the memory block for `native_code` must furthermore have the appropriate privileges to execute machine instructions. See for details.

See also:     amx_Init

---

**amx_MemInfo**                          Return memory size information

Syntax:     `int AMXAPI amx_MemInfo(AMX *amx, long *codesize,`
                       `long *datasize, long *stackheap)`

      `amx`          The abstract machine.

      `codesize`     Will hold the size of the executable code plus the code header upon return. See appendix D for a description of the header.

      `datasize`     Will hold the size of the global/static data upon return.

      `stackheap`    Will hold the combined (maximum) size of the of the stack and the heap upon return.

Notes:      All sizes are in bytes.

The stack and the heap share a memory region; the stack grows towards the heap and the heap grows towards the stack.

---

**amx_NameLength**                          Return the maximum name length

Syntax:     `int amx_NameLength(AMX *amx, int *length)`

      `amx`          The abstract machine.

      `length`       Will hold the maximum name length upon return. The returned value includes the space needed for the terminating zero byte.

See also:     amx_GetPublic, amx_GetPubVar

---

**amx_NativeInfo**                              Return a structure for amx_Register

Syntax:          `AMX_NATIVE_INFO *amx_NativeInfo(char *name,`
                                           `AMX_NATIVE func)`

        `name`           The name of the function (as known to the PAWN program)

        `func`           A pointer to the native function.

Returns:         A pointer to a static record (this record is overwritten on every call; it is *not* thread-safe).

Notes:           This function creates a list with a single record for **amx_Register**. To register a single function, use the code snippet (where `my_solve` is a native function):

```
err = amx_Register(amx, amx_NativeInfo("solve", my_solve), 1);
```

See also:        **amx_Register**

---

**amx_NumNatives**                              Return the number of native functions

Syntax:          `int amx_NumNatives(AMX *amx, int *number)`

        `amx`            The abstract machine.

        `number`         Will hold the number of native functions upon return.

Notes:           The function returns number of entries in the file's "native functions" table. This table holds only the native functions that the script refers to (i.e. the function that it calls). To retrieve the function names, use **amx_GetNative**.

See also:        **amx_GetNative**, **amx_NumPublics**

## amx_NumPublics · Return the number of public functions

Syntax:     `int amx_NumPublics(AMX *amx, int *number)`

`amx`         The abstract machine.

`number`    Will hold the number of public functions upon return.

Notes:     The function returns number of entries in the file's "public functions" table. To retrieve the function names, use `amx_GetPublic`.

See also:   `amx_GetPublic`, `amx_NumPubVars`

## amx_NumPubVars · Return the number of public variables

Syntax:     `int amx_NumPubVars(AMX *amx, int *number)`

`amx`         The abstract machine.

`number`    Will hold the number of public variables upon return.

Notes:     The function returns number of entries in the file's "public variables" table. To retrieve the variable names, use `amx_GetPubVar`.

See also:   `amx_GetPubVar`, `amx_NumPublics`

## amx_Push · Pass a numeric argument "by-value"

Syntax:     `int amx_Push(AMX *amx, cell value)`

`amx`         The abstract machine.

`value`      The value to pass to the public function.

Notes:     Any parameters to a public function must be pushed to the function before calling `amx_Exec`. If a public function has multiple arguments, the arguments must be pushed *int reverse order*.

See also:   `amx_Exec`, `amx_PushAddress`, `amx_PushArray`, `amx_PushString`

## amx_PushAddress                    Pass a numeric argument with relocation

Syntax:        `int amx_PushAddress(AMX *amx, cell *value)`

                `amx`           The abstract machine.

                `value`         The value to pass to the public function.

Notes:         This function behaves like `amx_Push`, but it first relocates the value from the "host address space" (such as what the C/C++ program uses) to the virtual address space of the abstract machine. This relocation is needed to pass an address that was allocated with `amx_Allot` to a public function.

See also:      `amx_Allot`, `amx_Exec`, `amx_Push`

## amx_PushArray                    Pass an argument or array "by-reference"

Syntax:        `int amx_PushArray(AMX *amx, cell **address,`
                                  `const cell array[], int numcells)`

        `amx`          The abstract machine.

        `address`    The address of the cell for C/C++ programs to access. This parameter may be `NULL`.

        `array`       The array of values to pass to the public function. A single cell that must be passed *by-reference* is regarded as a single-cell array.

        `numcells`   The number of elements in the array.

Notes:         Any parameters to a public function must be pushed to the function before calling `amx_Exec`. If a public function has multiple arguments, the arguments must be pushed *int reverse order*.

                The function allocates memory for the array inside the "heap" of the abstract machine. This memory must be freed with `amx_Release` (passing in parameter `address`).

See also:      `amx_Exec`, `amx_Push`, `amx_PushString`, `amx_Release`

## amx_PushString                                      Pass a string argument

Syntax:     `int amx_PushString(AMX *amx, cell **address,`
            `                   const char *string, int pack,`
            `                   int use_wchar)`

| | |
|---|---|
| `amx` | The abstract machine. |
| `address` | The address of the cell for C/C$^{++}$ programs to access. This parameter may be `NULL`. |
| `string` | The string to pass to the public function. |
| `pack` | Non-zero to convert the source string to a packed string in the abstract machine, zero to convert the source string to a cell string. |
| `use_wchar` | A non-zero value interprets the `string` argument as a pointer to "wide characters" —i.e. `wchar_t`, regardless of its `char` type. This allows the function to accept Unicode strings. |

Notes:      Any parameters to a public function must be pushed to the function before calling `amx_Exec`. If a public function has multiple arguments, the arguments must be pushed *int reverse order*.

The function allocates memory for the array inside the "heap" of the abstract machine. This memory must be freed with `amx_Release` (passing in parameter `address`).

When you pass in an Unicode string and request a packed format in the abstract machine (i.e. both `pack` and `use_wchar` are *true*), the characters are truncated to 8-bits.

See also:   `amx_Exec`, `amx_Push`, `amx_PushArray`, `amx_Release`, `amx_SetString`

---

## amx_RaiseError                                                Flag an error

---

Syntax:        int amx_RaiseError(AMX *amx, int error)

              amx            The abstract machine.

              error          The error code. This is the code that amx_Exec re-
                                 turns.

Notes:         This function should be called from a native function. It lets the
              default callback routine return an error code.

---

## amx_Register                                      Make native functions known

---

Syntax:        int amx_Register(AMX *amx, AMX_NATIVE_INFO *list,
                                 int number)

              amx            The abstract machine.

              list           An array with structures where each structure holds
                                 a pointer to the name of a native function and a func-
                                 tion pointer. The list is optionally terminated with a
                                 structure holding two NULL pointers.

              number         The number of structures in the list array, or -1 if the
                                 list ends with a structure holding two NULL pointers.

Notes:         On success, this function returns 0 (AMX_ERR_NONE). If this func-
              tion returns the error code AMX_ERR_NOTFOUND, one or more native
              functions that are used by the PAWN program are not found in the
              provided list. You can call amx_Register again to register addi-
              tional function lists.

              To check whether all native functions used in the compiled script
              have been registered, call amx_Register with the parameter list
              set to NULL. This call will not register any new native functions, but
              still return AMX_ERR_NOTFOUND if any native function is unregistered.

See also:      amx_NativeInfo

## amx_Release            Free heap space in the abstract machine

Syntax:      `int amx_Release(AMX *amx, cell *address)`

        `amx`        The abstract machine.

        `address`    The address of the allocated cell as the PAWN program (that runs in the abstract machine) sees it. This value is returned by `amx_Allot`, `amx_PushArray` and `amx_PushString`.

Notes:     `amx_Allot` allocates memory on the heap in ascending order (the heap grows upwards). `amx_Release` frees all memory *above* the value of the input parameter `address`. That is, a single call to `amx_Release` can free multiple calls to `amx_Allot` if you pass the `address` value of the first allocation.

            `amx_PushArray` and `amx_PushString` use `amx_Allot` internally, so the same procedure applies to these functions as well.

See also:    `amx_Allot`, `amx_PushArray`, `amx_PushString`

## amx_SetCallback            Install a callback routine

Syntax:      `int amx_SetCallback(AMX *amx, AMX_CALLBACK callback)`

        `amx`        The abstract machine.

        `callback`   The address for a callback function. See function `amx_Callback` for the prototype and calling convention of a callback routine.

Notes:     If you change the callback function, you should not use functions `amx_Register` or `amx_RaiseError`. These functions work in combination with the default callback function. To set the default callback, set parameter `callback` to the function `amx_Callback`.

            You may set the callback before or after calling `amx_Init`.

## amx_SetDebugHook                                    Install a debug routine

Syntax:     `int amx_SetDebugHook(AMX *amx, AMX_DEBUG debug)`

amx         The abstract machine.

debug       The address for a callback function for the debugger.
            The prototype and calling convention of a debug hook
            routine are:

```
int AMXAPI CallbackFunction(AMX *amx);
```

Notes:      To disable the debug hook, set the `debug` parameter to `NULL`.

## amx_SetString                          Store a string in the abstract machine

Syntax:     `int amx_SetString(cell *dest, char *source, int pack,`
                        `int use_wchar, size_t size)`

dest        A pointer to a character array in the AMX where the
            converted string is stored.

source      A pointer to the source string.

pack        Non-zero to convert the source string to a packed
            string in the abstract machine, zero to convert the
            source string to a cell string.

use_wchar   A non-zero value interprets the `source` argument as
            a pointer to "wide characters" —i.e. `wchar_t`, regard-
            less of its `char` type. This allows the function to ac-
            cept Unicode strings.

size        The maximum number of *cells* to store in `dest`, in-
            cluding the terminating zero byte or cell. If the string
            in the `source` is longer than can fit in the number of
            cells in `dest`, it will be truncated.

Notes:      When you pass in an Unicode string and request a packed format in
            the abstract machine (i.e. both `pack` and `use_wchar` are *true*), the
            characters are truncated to 8-bits.

See also:   amx_GetString

## amx_SetUserData                                        Set general purpose user data

Syntax:        `int amx_SetUserData(AMX *amx, long tag, void *ptr)`

  `amx`         The abstract machine.

  `tag`         The "tag" of the user data, which uniquely identifies
                the user data. This value should not be zero.

  `ptr`         A pointer to the user data.

Notes:         The AMX stores multiple "user data" fields. Each field must have
               a unique tag. The tag may be any value (as long as it is unique)
               except zero, but it is usually formed by four characters through the
               macro `AMX_USERTAG`.

```
r = amx_SetUserData(amx, AMX_USERTAG('U','S','E','R'), "Fire");
```

               The AMX does not use "user data" in any way. The storage can be
               used for any purpose.

See also:      amx_GetUserData


## amx_StrLen                                        Get the string length in characters

Syntax:        `int amx_StrLen(const cell *cstring, int *length)`

  `cstring`     The string in the abstract machine.

  `length`      This parameter will hold the string length upon re-
                turn.

Notes:         This function determines the length in *characters* of the string, not
               including the zero-terminating character (or cell). A packed string
               occupies less cells than its number if characters.

               If the `cstring` parameter is `NULL`, the `length` parameter is set to
               zero (0) and the function returns with an error code.

               For converting unpacked strings to UTF-8, function amx_UTF8Len
               may be more convenient.

See also:      amx_GetString, amx_SetString, amx_StrParam, amx_UTF8Len

---

**amx_StrParam**                    Get a string parameter from an abstract machine

Syntax:          amx_StrParam([AMX*] amx, [int] param, [void*] result)

                 amx_StrParam_Type([AMX*] amx, [int] param, result, type)

     amx          The abstract machine.

     param        The parameter number.

     result       A the address of a variable (or a memory buffer) that will hold the result on return.

     type         On C compilers, a "void pointer" may be assigned the address of any buffer. With C++, however, the type of parameter result must be explicitly passed to the macro. That is, the type parameter must be char* or wchar_t* or similar.

Notes:           This **macro** allocates a block of memory and copies a string parameter into that block. See page 55 for an example of using this macro.

                 The macro uses variable-length arrays if available (i.e. on a C99 compiler), and alloca otherwise.

See also:        amx_GetString, amx_StrLen

---

**amx_UTF8Check**                    Check whether a string is valid UTF-8

Syntax:          int amx_UTF8Check(const char *string, int *length)

     string       A zero-terminated string.

     length       If the string is a valid UTF-8 string, this parameter will hold the length of the string (in wide characters, and excluding the terminating zero character). This parameter may be NULL in case you do not need the length.

Notes:    The function runs through a zero-terminated string and checks the validity of the UTF-8 encoding. The function returns an error code, it is `AMX_ERR_NONE` if the string is valid UTF-8 (or valid ASCII for that matter).

If the string is valid UTF-8, you can use the value of parameter `length` to allocate enough memory to hold a string that you can convert with amx\_UTF8Get.

See also:    amx\_UTF8Get, amx\_UTF8Put, amx\_UTF8Len

---

## amx\_UTF8Get                                    Decode a character from UTF-8

Syntax:    `int amx_UTF8Get(const char *string, const char **endptr,`
                    `cell *value)`

string    A pointer to the start of an UTF-8 encoded character.

endptr    This pointer will point to the UTF-8 character behind the one that is decoded after the function completes. As UTF-8 encoding is variable-length, this returned value is useful when decoding a full string character by character. This parameter may be `NULL`.

value    A pointer to the "wide" character that has the value of the decoded UTF-8 character. This parameter may be `NULL`.

Notes:    The function returns an error code. On error, `endptr` points to the start of the character (the same value as the input value for the `string` parameter) and `value` is set to zero.

See also:    amx\_UTF8Check, amx\_UTF8Put

---

**amx_UTF8Len**                    Return the length of the string in UTF-8 encoding

Syntax:      `int amx_UTF8Len(const cell *string, int *length)`

      `string`     A zero-terminated string. This should normally be an unpacked string.

      `length`     Upon return of the function, this parameter holds the number of bytes that are needed to store the string in UTF-8 encoding, *excluding* the zero terminator. If the input string is a packed string, the returned length is the same as the string length —packed strings should not be UTF-8 encoded.

Notes:       If the `cstring` parameter is `NULL`, the `length` parameter is set to zero (0) and the function returns with an error code.

See also:    amx_StrLen, amx_UTF8Check

---

**amx_UTF8Put**                              Encode a character into UTF-8

Syntax:      `int amx_UTF8Put(char *string, char **endptr,`
                `int maxchars, cell value)`

      `string`     A pointer to the string that will hold the UTF-8 encoded character. This parameter may *not* be `NULL`.

      `endptr`     This pointer will point directly *behind* the encoded UTF-8 character after the function completes. As UTF-8 encoding is variable-length, this returned value is useful when encoding a sequence of Unicode/UCS-4 characters into an UTF-8 encoded string. This parameter may be `NULL`.

      `maxchars`  The maximum number of characters that the function may use. An UTF-8 character is between 1 and 6 bytes long. If the character value in the parameter `value` is restricted to the Basic Multilingual Plane (16-bits Unicode), the encoded length is between 1 and 3 bytes.

      `value`      The "wide" character with the value to be encoded as an UTF-8 character.

Notes:    The function returns an error code if the parameter `maxchars` is lower than the required number of bytes for the UTF-8 encoding; in this case nothing is stored in the `string` parameter.

The function does not zero-terminate the string.

Character values that are invalid in Unicode/UCS-4 cannot be encoded in UTF-8 with this routine.

See also:    `amx_UTF8Check`, `amx_UTF8Get`

## Error codes (run-time errors)

`AMX_ERR_NONE`                                                          (0)
>       No error.

`AMX_ERR_EXIT`                                                          (1)
>       Program aborted execution. This is usually not an error.

`AMX_ERR_ASSERT`                                                        (2)
>       A run-time assertion failed.

`AMX_ERR_STACKERR`                                                      (3)
>       Stack or heap overflow; the stack collides with the heap.

`AMX_ERR_BOUNDS`                                                        (4)
>       Array index is out of bounds.

`AMX_ERR_MEMACCESS`                                                     (5)
>       Accessing memory that is not allocated for the program.

`AMX_ERR_INVINSTR`                                                      (6)
>       Invalid instruction.

`AMX_ERR_STACKLOW`                                                      (7)
>       Stack underflow; more items are popped off the stack than were pushed onto it.

`AMX_ERR_HEAPLOW`                                                       (8)
>       Heap underflow; more items are removed from the heap than were inserted into it.

`AMX_ERR_CALLBACK`                                                      (9)
>       There is no callback function, and the program called a native function.

`AMX_ERR_NATIVE`                                                       (10)
>       Native function requested the abortion of the abstract machine.

`AMX_ERR_DIVIDE`                                                       (11)
>       Division by zero.

`AMX_ERR_SLEEP` (12)

The script, or a native function, forced a "sleep". A host application may implement a simple kind of co-operative multitasking scheme with the "sleep" instruction.

`AMX_ERR_INVSTATE` (13)

A function was called in a state for which the function is undefined (and there is no fall-back function).

`AMX_ERR_MEMORY` (16)

General purpose out-of-memory error.

`AMX_ERR_FORMAT` (17)

Invalid format of the memory image for the abstract machine.

`AMX_ERR_VERSION` (18)

This program requires a newer version of the abstract machine.

`AMX_ERR_NOTFOUND` (19)

The requested native functions are not found.

`AMX_ERR_INDEX` (20)

Invalid index (invalid parameter to a function).

`AMX_ERR_DEBUG` (21)

The debugger cannot run (this is an error code that the debug hook may return).

`AMX_ERR_INIT` (22)

The abstract machine was not initialized, or it was attempted to double-initialize it.

`AMX_ERR_USERDATA` (23)

Unable to set user data field (table full), or unable to retrieve the user data (not found).

`AMX_ERR_INIT_JIT` (24)

The Just-In-Time compiler failed to initialize.

`AMX_ERR_PARAMS` (25)

General purpose parameter error: one of the parameters to a function of the abstract machine was incorrect (e.g. out of range).

`AMX_ERR_DOMAIN` (26)

A "domain error": the expression result does not fit in the variable that must hold it. This error may occur in fixed point and floating point support libraries.

# Extension modules

An extension module provides a PAWN program with application-specific ("native") functions. An native function is a function that is implemented in the host application (as opposed to being implemented in the PAWN script) and it is typically implemented in a different programming language.* Creating an extension module is a three-step process:

1 writing the native functions (in C);
2 making the functions known to the abstract machine;
3 writing an include file that declares the native functions for the PAWN programs.

## • 1. Writing the native functions

Every native function must have the following prototype:

```
cell AMX_NATIVE_CALL func(AMX *amx, const cell *params);
```

The identifier "`func`" is a placeholder for a name of your choice. The `AMX` type is a structure that holds all information on the current state of the abstract machine (registers, stack, etc.); it is defined in the include file `AMX.H`. The symbol `AMX_NATIVE_CALL` holds the calling convention for the function. The file `AMX.H` defines it as an empty macro (so the default calling convention is used), but some operating systems or environments require a different calling convention. You can change the calling convention either by editing `AMX.H` or by defining the `AMX_NATIVE_CALL` macro before including `AMX.H`. Common calling conventions are `_cdecl`, `_far _pascal` and `_stdcall`.

The `params` argument points to an array that holds the parameter list of the function. The value of `params[0]` is the number of *bytes* passed to the function (divide by the size of a `cell` to get the number of parameters passed to the function); `params[1]` is the first argument, and so forth.

For arguments that are passed by value, the respective element in the `params` arrays is the value. When an argument is passed by reference, the respective element in the `params` array contains an address. A native function can use this address to access variables inside the abstract machine directly.

---

* The native function interface is technically known as a "foreign function interface", but this manual uses the term "native function interface".

When a native function accepts a variable number of arguments, all arguments in the "variable argument list" are passed to the native function by reference. Even (literal) constants that are passed to the function are first stored on a temporary location on the stack and then the address of that location is passed to the function —the constant is thereby passed "by reference".

Strings, like other arrays, are always passed by reference. However, neither packed strings nor unpacked strings are universally compatible with C strings.[†] Therefore, the abstract machine API provides two functions to convert C strings to and from PAWN strings: amx_GetString and amx_SetString.

A native function may abort a program by calling amx_RaiseError with a non-zero code. The non-zero code is what amx_Exec returns.

### • 2. Linking the functions to the abstract machine

An application uses amx_Register to make any native functions known to the abstract machine. Function amx_Register expects a list of AMX_NATIVE_INFO structures. Each structure holds a pointer to the name of the native function and a function pointer.

Below is a full example of a file that implements two simple native functions: raising a value to a power and calculating the square root of a value. The list of AMX_NATIVE_INFO structures is near the bottom of the example —it is wrapped in an "initialization function" called amx_PowerInit.

```
/* This file implements two the native functions: power(value,exponent)
 * and sqroot(value).
 */
#include "amx.h"

static cell n_power(AMX *amx, const cell *params)
{
  /* power(value, exponent);
   *    params[1] = value
   *    params[2] = exponent
   */
  cell result = 1;
  cell exponent = params[2];
  while (exponent-- > 0)
    result *= params[1];
```

---

[†] On Big Endian computers, packed strings are compatible with C strings; when the wchar_t type is the size of a cell, unpacked strings are compatible with Unicode strings.

```
    return result;
}

static cell n_sqroot(AMX *amx, const cell *params)
{
  /* sqroot(value);
   *   params[1] = value
   * This routine uses a simple successice approximation algorithm.
   */
  cell div = params[1];
  cell result = 1;
  while (div > result) {          /* end when div == result, or just below */
    div = (div + result) / 2;   /* take mean value as new divisor */
    result = params[1] / div;
  } /* while */
  return div;
}

int amx_PowerInit(AMX *amx)
{
  static AMX_NATIVE_INFO power_Natives[] = {
    { "power",  n_power },
    { "sqroot", n_sqroot },
    { 0, 0 }        /* terminator */
  };
  return amx_Register(amx, power_Natives, -1);
}

int amx_PowerCleanup(AMX *amx)
{
  return AMX_ERR_NONE;
}
```

In your application, you must add a call to `amx_InitPower` with the "`amx`" structure as a parameter, as shown below:

```
err = amx_InitPower(&amx);
```

The first example of "host applications" for the PAWN abstract machine called `amx_Register` directly, referring to the external arrays `core_Natives` and `console_Natives` (being the native function tables). In many situations, the strategy taken here (calling a function provided by the extension module to handle the native function registration) is preferable:

◇ Giving a function "external" scope is safer than doing so with a variable; as opposed to functions, variables can be (accidentally) tampered with. Observe, by the way, that only the functions `amx_PowerInit` and `amx_PowerCleanup` have external scope in the above example.

◇ An extension module may require additional "start-up" code. Doing this in the same routine that also registers the native functions makes sure that all initialization steps occur, and in the correct order.

◇ An extension module may also require clean-up code. When all extension modules provide "initialization" and "clean-up" functions, the rules for adding an extension module to the host application become universal. This is especially so if there is a naming convention for these initialization and clean-up functions. For this reason, even though the "power" extension module does not require any clean-up, an empty clean-up function `amx_PowerCleanup` was added.

### • 3. writing an include file for the native functions

The first step implements the native functions and the second step makes the functions known to the abstract machine. Now the third step is to make the native functions known to the PAWN compiler. To that end, one writes an include file that contains the prototypes of the native functions and all constants that may be useful in relation to the native functions.

```
#pragma library Power
native power(value, exponent)
native sqroot(value)
```

The `#pragma library` line is useful when you create a dynamically loadable extension module, as described on page 60; it is not required for an extension module that is statically linked to the host application.

## Writing "wrappers"

The preceding sections described the implementation of a few functions that were specially crafted as "native functions" for the PAWN abstract machine. It is common practice, however, that instead of writing *new* functions for PAWN, you will make a set of existing C/C++ functions available to PAWN. To "glue" the existing functions to PAWN, you need to embed each function in a tiny new function with the required "native function" signature. Such new functions are called wrapper functions.

Wrapper functions also illustrate the issues in passing parameters across C/C++– PAWN boundaries, plus that they provide templates for writing any kind of native functions.

### • Pass-by-value, the simplest case

The PAWN toolset was designed to make the interface to native functions quick and easy. To start with an example, I will make a wrapper for the function `isalpha` from the standard C library. The prototype for `isalpha` is:

```
    int isalpha(int c);
```

Wrapping `isalpha` into a native function, results in the code:

```
static cell n_isalpha(AMX *amx, const cell *params)
{
  return  isalpha( (int)params[1] );
}
```

In addition to writing the above wrapper function, you must also still add it to a table for `amx_Register` and add it to an include file for the PAWN compiler.

## • Floating point

Wrapping functions like `isalpha` represent the simplest case: functions that take parameters with an "integer" type and that return "void" or an integer type. When either any of the parameters or the return type of the existing function are a floating point type, these parameters must be cast to/from the "`cell`" type that PAWN uses —but this cast must happen through a special macro. For example, consider the function `sin` with the prototype:

```
    double sin(double angle);
```

Its wrapper function is:

```
static cell n_sin(AMX *amx, const cell *params)
{
  float r =  sin( amx_ctof(params[1]) );
  return amx_ftoc(r);
}
```

The symbols `amx_ctof` and `amx_ftoc` are *macros* that cast a "`cell`" type into "`float`" (or "`double`", depending on the cell size) and vice versa, but in contrast to the standard type casts of C/C$^{++}$ they do not change the bit representation of the value that is cast. A normal type cast, therefore, changes the value[*] and what is needed is a cast that leaves the value intact —which is what `amx_ctof` and `amx_ftoc` do.

---

[*]  This behaviour is quite apparent in the cast from floating point to integer, which *truncates* the value to its integral part.

### • Strings

Wrapping functions that take string parameters is more involved, because the memory layout of a string in the PAWN abstract machine is probably different than that of C/C++.[†] This means that strings must be converted between the native (wrapper) function and the PAWN abstract machine. The standard C function `access` has the prototype:

```
int access(const char *filename, int flags);
```

Its wrapper function is:

```
static cell n_access(AMX *amx, const cell *params)
{
  int r = 0, length;
  char *pname;

  amx_StrLen((cell*)params[1], &length);
  if ((pname = malloc(length + 1)) != NULL) {
    amx_GetString(pname, (cell*)params[1], 0, length + 1);
    r =  access( pname, (int)params[2] );
    free(pname);
  } /* if */
  return r;
}
```

When the PAWN abstract machine passes an array to a native function, it passes the base address of that array in the respective field of `params`. For the functions `amx_StrLen` and `amx_GetString`, this field must be cast to a "cell*" type. In the above snippet, this is done with an explicit cast, but it is advised to use the macro `amx_Address` to do the cast. Actually, if the size of a "cell" of the PAWN abstract machine is smaller than a pointer, the address in the `params` field must be expanded to a full address, and the macro `amx_Address` does this too —if needed. For example, when a PAWN cell is 32-bit in a 64-bit host application, `amx_Address` converts 32-bit truncated addresses to full 64-bit addresses. Using the macro `amx_Address` therefore increases portability, but it is not strictly needed in the typical case that the pointer size and cell size are equal.

The next step is to convert the string for the format as it is stored in the abstract machine to what C/C++ understands. Function `amx_GetString` does that, but before using it, you have to check the string length first —hence, `amx_StrLen`. The last parameter of `amx_GetString` also allows you to limit the number of characters

---

[†]  On a Big Endian CPU platform *packed* strings have the same memory layout in Pawn and in C/C++, *unpacked* strings and all strings on a Little Endian CPU have a different layout.

stored in the destination. Function `amx_GetString` recognizes both packed and unpacked strings, by the way.

If you need to write a string back into the data section of the abstract machine, you can use the `amx_SetString` companion function.

When making wrappers by hand, the macro `amx_StrParam` may be convenient because it implements the "scaffolding code". The wrapper for the function `access` would become:

```
static cell n_access(AMX *amx, const cell *params)
{
  int r = 0;
  char *pname;

  amx_StrParam(amx, params[1], pname);
  if (pname != NULL)
    r =  access(  pname, (int)params[2] );
  return r;
}
```

The wrapper function uses the C function `alloca` to allocate memory, instead of `malloc`. The advantage of `alloca` is that memory does not need to be freed explicitly. Function `alloca` is not in the ANSI C standard, however, and it may not be available on your platform.

### • Pass-by-reference

C/C++ functions that return values through pointers need a similar wrapping as strings: PAWN does not understand pointers, but it supports call-by-reference. The example function for this wrapper is the C/C++ function `time`, with prototype:

```
time_t time(time_t* timer);
```

I am making the bold assumption that `time_t` is represented as a 32-bit integer (which as `cell` is as well). The wrapper function becomes:

```
static cell n_time(AMX *amx, const cell *params)
{
  time_t r;
  cell *cptr;

  assert(sizeof(cell) == sizeof(time_t));
  cptr = amx_Address(amx, params[1]);
  r =  time(  (time_t*)cptr );
  return r;
}
```

In the above wrapper function, function `time` writes directly into a memory `cell` in the data section of the abstract machine. This is allowed only if the value that the function writes has the same size as a `cell` (32-bit). For good measure, the above wrapper verifies this with an `assert` statement. If the size that the C/C++ function returns differs from that of a `cell`, the wrapper function must convert it to a `cell` before writing it through the pointer obtained by `amx_Address`.

## • Arrays

For the interface of the abstract machine to the host application, a "reference parameter" (see the preceding section) is identical to an array with one element. Writing wrappers for functions that take an array is therefore similar to writing a function that handles a reference argument. With single dimensional arrays, the main difference is that the pointer in the respective `params` element now points to the first cell of potentially many cells.

Multi-dimensional arrays must be handled differently, though, as the memory lay-out differs between C/C++ and PAWN. In comparison with C/C++, two-dimensional arrays in PAWN are *prefixed* with a single-dimensional array that holds memory offsets to the start of each "row" in the two-dimensional array. This extra list allows each row to have a different column length. In C/C++, each column in a two-dimensional array must have the same size.

If you are writing a wrapper function for an existing C function, as opposed to writing/adapting a native function specifically to exploit PAWN's features, you will not be concerned with variable column-length arrays —C/C++ does not support them, so your native function will not allow them. All that needs to be done, then, is to skip the prefixed "column offsets" list after getting the address from the `params` array.

For an example, I use the OpenGL function `glMultMatrixf` which multiplies a given $4 \times 4$ matrix with the *current* matrix. The prototype of the function is:

```
void glMultMatrixf(const GLfloat  *m);
```

The wrapper function just has to get the address of its array parameter and add four cells to them.

```
static cell n_glMultMatrixf(AMX *amx, const cell *params)
{
  cell *cptr;
```

```
        assert(sizeof(cell) == sizeof(time_t));
        cptr = amx_Address(amx, params[1]);
         glMultMatrixf(  (GLfloat*)(cptr + 4) );
        return 0;
    }
```

For this example, I selected the OpenGL matrix multiplication function that accepts a matrix of "`float`-type" floating point values, because the `cell` and the `float` types are both four bytes (in a common PAWN implementation). If you need to wrap a function that accepts an array of "`double`-type" values, this array has to be converted from `float` to `double` values —and possibly back to `float` after calling the wrapped function.

## • Wrapping class methods (C++ interface)

The interface between the abstract machine and C/C++ is based on plain functions. When trying to use a *class method* as a native function, there is a complexity: a (non-static) class method function must be called with an implicit "`this`" parameter, which the abstract machine is unaware of. Hence, the abstract machine cannot pass this parameter directly and some extra intermediate code is needed to call a class method.

Reasons why you wish to use class methods as native functions, rather than plain C/C++ functions are:
1. improved encapsulation,
2. or the ability to bind a different *instance* of the class to each abstract machine (when several abstract machines exists concurrently).

In the first case, declaring the class methods and member variables as "`static`" is a solution. Static methods do not receive a `this` parameter, but, in turn, they cannot access non-static member variables. So the member variables should be static too.

This section covers the second case: binding an abstract machine to a class instance that is created dynamically. For this binding, the interface needs "forwarding" functions that call the appropriate (non-static) class method and a look-up mechanism to match the required `this` to the abstract machine. The forwarding functions might be static methods in the same class. The example below, however, uses plain functions to wrap a C++ class without modifying the class.

The wrapper is for an imaginary class that allows writing to "log files". With this procedure, each abstract machine will get its own log file. For purpose of showing the wrapper, the class is kept rather simplistic:

```
class LogFile {
  FILE *f;

public:
  LogFile()
  {
    f = tmpfile();
  }

  ~LogFile()
  {
    fclose(f);
  }

  bool write(char *string)
  {
    int  r = fprintf(f, "%s\n", string);
    return r > 0;
  }
};
```

When a new abstract machine initializes its "log file" native functions, it must create a new instance of the class and *bind* the instance (the `this` pointer) to the abstract machine. Later, the wrapping/forwarding function must have a way to look up this binding —or a way to find the `LogFile` class instance attached to the abstract machine. The simplest way to implement this binding is to store a pointer to the class instance in the "user data" of the abstract machine. However, as the number of user values for an abstract machine is limited, this is not a general purpose solution: if every extension module (string functions, console functions, date/time functions, etc) needs a user value, you'll run out quickly. An alternative simple method that keeps the binding local to the extension module is the use of the `map` container class from the Standard Template Library (STL). The STL is now part of the C++ standard library, so it is likely to be available on your system.

User data example: 13

```
static std::map<AMX*, LogFile*> LogFileLookup;

static cell n_write(AMX* amx, cell params[])
{
  int r = 0;
  char *pstr;

  amx_StrParam_Type(amx, params[1], pstr, char*);
  std::map<AMX*, LogFile*>::iterator p = LogFileLookup.find(amx);
  if (pstr != NULL && p != LogFileLookup.end())
    r = p->second->write(pstr);
```

```
    return r;
}

extern "C"
int amx_LogFileInit(AMX* amx)
{
  LogFile* lf = new LogFile;
  if (lf) {
    LogFileLookup.insert(std::make_pair(amx, lf));

    static AMX_NATIVE_INFO nativelist[] = {
      { "write",  n_write },
      { 0, 0 }        /* terminator */
    };
    return amx_Register(amx, nativelist, -1);
  } /* if */
  return AMX_ERR_MEMORY;
}

extern "C"
int amx_LogFileCleanup(AMX* amx)
{
  std::map<AMX*, LogFile*>::iterator p = LogFileLookup.find(amx);
  if (p != LogFileLookup.end()) {
    delete p->second;
    LogFileLookup.erase(p);
  } /* if */
  return AMX_ERR_NONE;
}
```

The wrapper function `n_write` contains the usual code to fetch a string parameter from the abstract machine (see page 55), and it also looks up the `LogFile` class instance for the abstract machine using the `map` container `LogFileLookup`. The function `amx_LogFileInit` creates the new instance and adds it to the map, in addition to registering native functions. The "clean up" function for the extension module does the reverse: it deletes the class instance and removes it from the map. Note that the `amx_LogFileInit` and `amx_LogFileCleanup` functions must be declared "`extern "C"`" (but the wrapper function `n_write` need not be).

The `map` container from the Standard Template Library is a general purpose implementation with a fair performance for very small to very large maps. From the description of the properties of the map, it appears that it uses an auto-balancing binary tree data structure. If you do not know (or do not control) how many abstracts machines can run concurrently, the STL `map` may be a good choice. On the other hand, if you can make an estimate of the typical number and/or the maximum number of concurrent abstract machines, you can typically improve the performance of the look-up by using a data structure that is tailored to the task and environment. Especially, a hash table can give a nearly constant

look-up time —meaning that looking up a class instance is equally quick when there are many concurrent abstract machines as when there are only few. The performance of a hash table deteriorates quickly when the table fills up, however, and very large hash tables have a bad overall performance because they do not fit in processor or disk caches.

## Dynamically loadable extension modules

Up to this point, the description for developing extension modules assumed static linking for the modules. This means that the object code for the modules is embedded in the same executable program/shared library as the rest of the host application. Static linking also means that if you wish to add more native functions, or correct a bug in one of the existing native functions, you need access to the source code of the host application.

The alternative is to build the extension module as a DLL (for Microsoft Windows) or in a shared library (for UNIX/Linux). When set up correctly, `amx_Init` will automatically load a dynamically loadable extension module and register its functions. When done, `amx_Cleanup`, cleans up the extension module and unloads it from the operating system.

Apart from freeing you from writing a few lines (you do not have to call the `amx_`*ModuleName*`Init` and `amx_`*ModuleName*`Cleanup` functions), the prime advantage of dynamic loading is that it makes the scripting subsystem of the host application easily extensible with "plug-in" extension modules. All that an end-user has to do to extend the scripting environment is to create or download a new extension module as a DLL/shared library, and to copy it with the associated include file (for the PAWN compiler) to appropriate (system) directories.

To build extension modules for dynamic loading, adhere to the following rules:

◇ Add a `#pragma library ...` line to the include file for the PAWN compiler. The PAWN compiler uses this `#pragma` to record which extension modules are actually referred to from the script. The PAWN compiler is smart enough to *avoid* including an extension module if the script does not call any of the functions in that extension module.

◇ The name of the DLL or shared library must be the same name as the one mentioned in the `#pragma library` line, but prefixed with the letters "amx" and with the extension " .dll" or ".so", whichever is appropriate.

◇ The extension module must at least provide the external/exported function `amx_`*Filename*`Init`, where *Filename* is, again, the name cited at the `#pragma library` line. If the library requires clean-up code, it should also provide the function `amx_`*Filename*`Cleanup`.

For example, when creating the example extension module "Power" from as a Windows DLL:
- the filename must be "`amxPower.dll`";
- the initialization and clean-up functions are must be named `amx_PowerInit` and `amx_PowerCleanup` respectively (that said, a do-nothing routine like `amx_PowerCleanup` may also be omitted);
- and the include file has the line "`#pragma library Power`" near the top —see also .

◇ Note that function names are case sensitive (and on Linux, filenames as well).

Please consult you compiler documentation for details for creating a DLL or a shared library; also look at B for details in building a dynamically loadable extension module, specifically to the section at .

◇ For deployment under UNIX/Linux, see also for an environment variable that you may need to set.

The flexibility of dynamically loadable extension modules is also the main reason why you may want to disable this feature: in the interest of *security*. If all native functions for your host application are carefully and selectively implemented by you, you have a good grip on what parts of the host application and of the operating system the end users can access. With "plug-in" extension modules, the entire system is effectively open, just as with any plug-in system.

To disable support for dynamically loadable extension modules, compile the abstract machine with the macro `AMX_NODYNALOAD` defined, see appendix B.

## Error checking in native functions

When comparing the wrapper functions for PAWN with those for other scripting languages, you may remark that the wrapper functions for PAWN are relatively small and easy. Notably, PAWN wrapper functions lack type and parameter checking that other scripting languages mandate. The wrapper function for `isalpha`, for example, does not check the number of parameters that the PAWN script passes in. The wrapper function *could have* check this number of arguments, because

isalpha() wrapper:

PAWN passes the number of bytes that the native function receives in `params[0]`, but in most cases this extra checking is redundant.

The number of parameters that are passed to a native function, and their tags, should be checked at compile-time, rather than at run-time. Therefore, PAWN requires the definitions of the native functions (in PAWN syntax), in addition to the implementation —this was the third step in the list at the start of the chapter "Extension modules" (page 49).

It is important that the native function declarations (in an include file) are accurate, and as specific as possible. For example, the native function declaration for the function `glMultMatrixf` would be:

```
native glMultMatrixf(const Float: m[4][4]);
```

The above declaration declares "`m`" as a $4 \times 4$ array, holding values that must have the "`Float`" tag. The PAWN compiler will now issue an error if a script passes a parameter to the function that is not a $4 \times 4$ array or that does not hold floating point values.

## Customizing the native function dispatcher

The above three steps to link native functions to the abstract machine imply that you use the default native function dispatcher. The default dispatcher is flexible and it has low overhead, but for specific purposes, you may create a *custom* native function dispatcher.

First, a little background. The abstract machine is much like a CPU implemented in software: it has an accumulator and a few other "registers", including an "instruction pointer" that points to the instruction that is executed next. When a function in a PAWN program calls some other function, the abstract machine sees a "CALL" instruction, which adjust the instruction so that the next instruction to be executed is the first instruction of the called function. So far, all is well. However, a native function cannot be called using the same procedure, as the native function is compiled for a *real* CPU and the abstract machine can only handle its own instruction set. A native function is not invoked with a "CALL" instruction, but with a "SYSREQ.C" instruction. Instead of adjusting the abstract machine's instruction pointer, a "SYSREQ.C" fires the native function dispatcher. The "SYSREQ.C" instruction could be compared with a software-invoked interrupt or trap.

It is the task of the native function dispatcher to find the correct native function, to call the function, and to return the function result. The prototype for a native function dispatcher is:

```
int amx_Callback(AMX *amx, cell index, cell *result, const cell *params);
```

where "`index`" is the unique identifier for the native function, "`params`" points to an array with parameters that the dispatcher should pass to the native function, and "`result`" is where the dispatcher should store the return value of the native function. Assuming that the native function dispatcher has a way of finding the appropriate native function from the `index`, the dispatcher can call the native function with:

```
*result = native_func(amx, params);
```

The default native function dispatcher works in combination with `amx_Register`, which looks up a function from the "native function table" in the header of the compiled program file and stores the physical function address directly in that table. With that done, the default dispatcher can simply use the `index` parameter as an index in the native function table and retrieve the physical address of the function. Several implementations of the default native function dispatcher go a step further: after looking up the address of the native function, the dispatcher changes the SYSREQ.C opcode into SYSREQ.D[*] and stores the function address as the parameter to SYSREQ.D. The result is that every next call to the native function will jump to the native function directly, without going through the native function dispatcher again.

This is a flexible scheme, as it allows you to inspect the compiled program and load only those packages with native functions that the program actually uses. It is also a scheme that imposes minimal overhead on calling native functions.

However, the standard operation of the tandem of `amx_Register` and the native function dispatcher assumes that the native function table can be adjusted at run-time. This is not the case when the script runs from ROM, for example. For those situations, you have the option of hard-coding the mapping of "SYSREQ" indices to native functions.

The first step to make is to adjust the declarations of native functions in the header files. Taking the example of the "power" function module, the new declarations become:

---

[*]  Turn to appendix D for details on the opcodes.

```
native power(value, exponent) = -1;
native sqroot(value)          = -2;
```

The difference with the declarations on page 52 is that the `power` function is now specifically set at "SYSREQ" −1 and `sqroot` is at "SYSREQ" −2. The use of negative numbers is mandatory: the PAWN compiler reserves positive numbers for its default auto-numbering scheme (both schemes can be mixed). When an explicit "SYSREQ" index is given for a native function, the PAWN compiler omits it from the native function table. That is, this scheme creates more compact binary files.

The default native function dispatcher needs help to map negative indices to function pointers; you need to create a table with the addresses of the native functions and then compile the abstract machine with the name of this table in a macro. In the table, the first entry is for index −1, the second for index −2, and so forth:

```
AMX_NATIVE my_natives[] = {
    n_power,    /* -1 */
    n_sqroot    /* -2 */
};
```

The second step is to compile the abstract machine with the following macro defined:

```
#define AMX_NATIVETABLE   my_natives
```

You can usually define the macro on the compiler's command line, for example with a syntax like "`-DAMX_NATIVETABLE=my_natives`".

If you need special functionality in the callback, an alternative is to replace the native function dispatcher with a custom version. This consists of two steps: creating the new native function dispatcher, and setting it. The latter is simply a matter of calling:

```
amx_SetCallback(&amx, my_callback);
```

An example of a native function dispatcher follows below —this version is equivalent to what you can do with the `AMX_NATIVETABLE` macro described above, by the way.

```
int my_callback(AMX *amx, cell index, cell *result, const cell *params)
{
  amx->error = AMX_ERR_NONE;
```

```
  switch (index) {
  case -1:
    *result = n_power(amx, params);
    break;
  case -2:
    *result = n_sqroot(amx, params);
    break;
  default:
    assert(0);
  } /* switch */

  return amx->error;
}
```

# Building the compiler

The C sources of the compiler conform mostly to ANSI C, as the toolset is intended to be widely portable. For purposes of distributing a multi-platform project file with the source code, I have settled on "CMake": a multi-platform tool for creating makefiles; see also appendix C. A CMake project file for the compiler is included in the "compiler" sub-directory where the PAWN source code is installed. There is another CMake project file that builds the abstract machine in the "amx" sub-directory.

If you cannot use CMake, you can create a "makefile" or a project for the sources with ease. To get started, type in a terminal or DOS box a command like:

```
cl sc1.c sc2.c sc3.c sc4.c sc5.c sc6.c sc7.c scexpand.c sci18n.c (...)
(...) sclist.c scmemfil.c scstate.c scvars.c lstring.c memfile.c
```

The "(...)" indicate a line continuation, by the way, you should not type these. The command `cl` stands for the compile-&-link utility for your compiler. This is indeed "`cl`" for Microsoft Visual C/C$^{++}$, "`bcc32`" for Borland C$^{++}$ and "`gcc`" for GNU GCC. The above command builds the PAWN compiler as a single executable, without a shared library/DLL. You can now customize the build by adding compiler options. on the command line.

The source code contains sections of code that are conditionally compiled. See your compiler manual how to specify options on the command line or in a "project" to set these options. The compiler source code also contains assertions to help me catch bugs while maintaining the code. To build the compiler without the assertions, compile the compiler with the NDEBUG definition set.

The CMake project file build the compiler as a shared library plus a tiny console-mode "driver" program that passes the command line options through to the library. You can also build the compiler as a static library, or as a stand-alone executable (as was explained in the preceding paragraphs). See for creating a static library.

When compiling the sources under Linux, you may need to first translate the CR/LF line endings to LF line endings —there are two source code archives for PAWN: the ZIP file has DOS/Windows-style line endings (CR/LF) and the TAR-GZIP file has UNIX-style line endings (LF). Some tools (e.g. some versions of the GCC compiler) are sensitive to the way that lines are ended. The utility "`dos2unix`" is the most convenient way to translate source files.

Note that the compiler uses an include file from the "amx" subdirectory too, so its probably best to run `dos2unix` over all source files in all subdirectories.

### • Compile-time options

The compiler is a stand-alone program. If you want to link it to an application, can compile the sources with the macro definition `NO_MAIN`. This will strip the "`main`" function and a set of I/O functions from the program. See the section "Embedding the compiler into an application" (below) for details.

If you want a PAWN compiler that outputs 16-bit P-code, add the definition "`PAWN_CELL_SIZE=16`" to the compiler options. Note that this is unrelated to whether the compiler itself is a 16-bit or a 32-bit executable program. The header file uses precise types for a compiler that conforms to the C99 standard, but for older (i.e. "most") compilers it boldly assumes that a "`short int`" is 16-bits and a "`long int`" is 32-bits. If this is not true for your compiler, you must change the definition of the cell type in `SC.H`, but you must also check the locations where sections that are conditionally compiled on the value of `PAWN_CELL_SIZE` appear. **N.B.** The PAWN tools are not regularly tested with 16-bit cells.

The basic code generation is followed by a simple peephole optimizer. If you stumble on a code generation bug, one of the first things that you may want to find out is whether this bug is in the code generation or in the optimizer. To do so, use the option `-d3` of the PAWN compiler (this replaces the `NO_OPTIMIZE` macro in previous releases to "conditionally compile" the peephole optimizer).

To save data space (which is important for the 16-bit version of the compiler, where data and stack must fit in one 64 kiB segment), two tables of strings are compressed; these tables are in `sc5.scp` and `sc7.scp`. If you change those strings (or add to them), the strings should be recompressed with the utility `SCPACK`. Before that, you have to build `SCPACK` itself —this is a simple ANSI C program with no dependencies on other files.

The PAWN compiler includes a preprocessor that does text substitutions (with or without parameters). The text matching capabilities of the PAWN preprocessor are even more flexible than that of the C/C$^{++}$ preprocessor, and, as a consequence, it is also at least as "dangerous" in obfuscating code. You may decide not to include the preprocessor (and the `#define` keyword) by setting the compile-time option `NO_DEFINE`.

The PAWN compiler reads source files in the ASCII character set and in the UTF-8 character set. Support for UTF-8 can be disabled by defining the macro

`PAWN_NO_UTF8`.  The UTF-8 decoder in PAWN supports the full 31-bit UCS-4 character set.

A few functions of the PAWN compiler are non-essential gadgets.  In cases where the size of the compiler counts, these can be removed by compiling with the `PAWN_LIGHT` macro defined.  With this macro defined, the compiler will miss:

◇ the usage report (cross-reference); i.e. the "`-r`" option,
◇ The stack/heap usage estimate, with the "`-d2`" and "`-d3`" options,
◇ the ability to parse response files; the "`@filename`" command line option is ignored,
◇ support for a PAWN.CFG file, whose options are implicitly read.
◇ generation of macro instructions and packed opcodes (i.e. the "`-O2`" option is disabled).

## • Summary of definitions

| | |
|---|---|
| `AMX_NO_MACRO_INSTR` | Removes the ability to generate supplemental & macro instructions from the PAWN compiler. Macro instructions are incompatible with a JIT run-time. |
| `AMX_NO_PACKED_OPC` | Removes the ability to generate packed opcodes from the PAWN compiler. Packed opcodes are incompatible with a JIT run-time. |
| `NDEBUG` | Compile without assertions. |
| `NO_MAIN` | Remove main() and I/O functions from the program. |
| `NO_DEFINE` | Remove the text preprocessor from the PAWN compiler (i.e. the `#define` directive). |
| `PAWN_CELL_SIZE` | The size of a cell in bits, either 16, 32 or 64. |
| `PAWN_LIGHT` | Remove support for cross-reference output and response files. Some optimizations are also disabled. This code implies `AMX_NO_MACRO_INSTR` and `AMX_NO_PACKED_OPC`. |
| `PAWN_NO_CODEPAGE` | Remove codepage support from the PAWN compiler. |
| `PAWN_NO_UTF8` | Remove the UTF-8 reading capability. |

## • Embedding the compiler into an application

When you want to link the PAWN compiler into an application, you will have to strip the "`main`" function from it (see the `NO_MAIN` option above). But that is just a first step. In addition, you should:

◇ Attend to the pollution of the global namespace by the many, many functions and global variables of the PAWN compiler.

⋄ Overrule the functions that the PAWN compiler calls for input/output.

The archive contains the file `libpawnc.c` which illustrates how to perform these steps. Basically, you implement all file I/O functions that the PAWN compiler requires. These functions do not have to read from file or write to file, you can compile from memory into memory, provided that you implement the functions that do this.

Then, from your application, call `pc_compile`, passing in all arguments. The prototype of the function is:

```
int pc_compile(int argc,char **argv)
```

As you can see, this function looks like the standard function `main`; when calling `pc_compile`, you must fill in an array of arguments, including `argv[0]` (because the compiler constructs the path for the include files from the path/filename in `argv[0]`).

Other functions that you can call from the application (before calling `pc_compile`) are `pc_addconstant` and `pc_addtag`. Function `pc_compile` removes all symbols before returning, including all constants and tagnames that you added with `pc_addconstant` and `pc_addtag`.

The LIBPAWNC.C file can also serve as the basis for a DLL or a shared library. As is, it can be used as a DLL/shared library for console applications —the `pc_error` function displays the error messages onto the console. Alternatively, you may add the "`-e`" option to the argument list of `pc_compile` to redirect all output to a file and use the LIBPAWNC library *without change* in GUI applications.

Compiling LIBPAWNC to a "shared library" (Linux, UNIX) is not much different than the compiling to a DLL, but you will probably want to reduce the symbols in the resulting library after the link step. The compiler sources export many functions that are only for internal use. The "strip" command allows you to strip all symbols from a library (shared or static), excluding a selected few (`pc_compile`, etc.). For a DLL, this step is unnecessary, because a DLL exports only symbols that were explicitly exported via a `.DEF` file.

Note that the exported functions in LIBPAWNC assume the default calling convention for the compiler. With many compilers, this is `__cdecl`. For a DLL, it is common to use `__stdcall`. You may be able to change the compiler's default calling convention with a (command line) option. However, some of the functions in the PAWN compiler use variable length argument lists, and your compiler may not provide support for variable length argument lists in the `__stdcall` calling

convention.*

The DLL version of LIBPAWNC can be driven from RUNDLL/RUNDLL32. The command line to use, for a 32-bit version, is:

```
rundll32 libpawnc.dll,Compile options hello.p
```

Among the recommended options are "`-D`" (set active directory for output and error files, "`-i`" (set the include path) and "`-e`" (send error messages to a file).

---

\*    It is widely believed that the ⎯stdcall calling convention does not allow variable length argument lists, but my reading of the specification suggests otherwise and I have successfully built ⎯stdcall functions that use variable length argument lists.

# Building the Abstract Machine

Project files to build the example "`pawnrun`" console run-time are available for Microsoft Visual C/C++ (in the "msvc" subdirectory) and for CMake. See the appendices A and C. for details. Mostly, though, you will want to embed the abstract machine in an application, instead of using a separate run-time. So the provided project and makefile are of limited use.

The library for the "Abstract Machine" (AMX) is fully implemented in a single C file: `amx.c`. This file contains the source code of all functions, but *without* any native function and *without* any optimizations. The key routine in the library, `amx_Exec`, is called the AMX *core* function, and it exists in various versions:

⋄ ANSI C: the slowest but most portable core (the default, requiring no extra files in addition to `amx.c`);

⋄ GNU GCC optimized: still implemented in C, but using specific GNU GCC extensions that make it significantly faster than the ANSI C version (Intel C/C++ also supports these extensions);

⋄ Intel Pentium assembler: this is a single design, but doubly implemented to support a wide range of assemblers;

⋄ ARM assembler (for architecture version 4 and up): this, too, is a single design, but doubly implemented to support the two most popular ARM assemblers;

⋄ Just-In-Time compilers: the fastest core (but the least portable, and currently only available for the Intel Pentium platform).

Next to the basic AMX library, the toolkit comes with various extension modules (native function libraries) that add console input/output, fixed point and floating point arithmetic, and helper routines to support the language. These extension modules are technically not part of the "Abstract Machine".

The C sources contain sections of code that are conditionally compiled. See your compiler manual how to specify options on the command line or in a "project" to set these options.

The source code of the AMX contains assertions to help me catch bugs while maintaining the code. In the retail version of the AMX, you will want to compile without assertions, because this code slows down its operation. To do so, compile the source files with the `NDEBUG` definition set.

The basic AMX library routines do not use or depend on dynamic memory allocation, file I/O or console I/O, but native functions may do so. For instance, the

"property" functions in the `AMXCORE.C` extension module use `malloc`/`free`; you can remove these property set/retrieval functions by compiling the `AMXCORE.C` file with the definition `AMX_NOPROPLIST`.

The console I/O functions in `AMXCONS.C` (another extension module) use standard C to a large extent. For a few extended functions, the file has explicit support for ANSI and VT100 terminal codes (`ANSI.SYS` under DOS, `xterm` and most shells under Linux), and for Win32 console programs. The `AMXCONS.C` file provides "hook" functions that your host application can implement to perform console output. By default, `AMXCONS.C` uses Win32 console functions when compiled for Microsoft Windows and ANSI/VT100 terminal codes when compiled for Linux or UNIX. If, on a Windows system, you prefer to use ANSI/VT100 terminal codes, compile with the macro `VT100`; if you wish to use your own "console I/O" functions, define `AMX_TERMINAL` instead —see the section "Adding a terminal to the abstract machine" on for examples.

Depending on the capabilities of the host application and the operating system, you may want to enable Unicode or "wide character" support for the scripting subsystem. The PAWN compiler is flexible in its handling of codepages and translation of extended ASCII and UTF-8 to wide characters (i.e., Unicode). For the host application, there are essentially two approaches:

1  Support Unicode or UCS-4 and interpret unpacked strings as strings holding "wide" characters. The PAWN compiler does *not* generate Unicode surrogate pairs. If characters outside the BMP ("Basic Multilingual Plane") are needed and the host application (or operating system) does not support the full UCS-4 encoding, the host application must split the 32-bit character `cell` provided by the PAWN compiler into a surrogate pair.
2  Support UTF-8 encoding and parse strings in the host application, or, if the operating system supports UTF-8 natively, pass the strings through to the higher level without further processing.

The core modules of the abstract machine are independent of whether the host application uses Unicode or UTF-8; the core modules of the abstract machine have wide character support built-in by default. Several auxiliary modules — for instance `AMXCONS.C` (console I/O support), need to be compiled with the `UNICODE` or `_UNICODE` macros defined to enable Unicode support. Both macros have the same effect. If you wish to remove Unicode/wide-character support, add the definition `AMX_ANSIONLY` to the compiler options. This option also removes UTF-8 support.

Calling conventions are always an important issue in porting software. The PAWN

AMX specifies the calling convention it uses via three macros. These macros are blank by default, in order to stay as close to ANSI C as possible. By (re-)defining either (or both) of these macros, you can adjust the calling conventions:

**AMX_NATIVE_CALL** — The calling convention for the native functions. You may want to set this to `__stdcall` when compiling for Win32.

**AMXAPI** — The calling convention used for all interface functions of the Abstract Machine (e.g. `amx_Init`), including the native function dispatcher and the the debugger callback. You need to change this if you put the AMX in a Windows DLL, for example.

**AMXEXPORT** — When you create dynamically loadable extension modules, the initialization and clean-up functions must be "visible" from the outside. For a UNIX/Linux shared library, any non-static function is automatically accessible, but for Microsoft Windows, a function must be explicitly exported. In addition, it is advised that exported functions use the `__stdcall` calling convention (i.e., set `AMXAPI` to `__stdcall`). See for details. A typical setting for `AMXEXPORT` under Microsoft Windows is `__declspec(dllexport)`.

If you intend to use the assembler core of the AMX, there are two more calling conventions to address.

As you may observe, the "calling convention" issue is a distinctive complexity of Microsoft Windows. In UNIX-like operating systems, you can usually ignore the issue of calling conventions.

The default threading method for the implementations for GNU C/C$^{++}$, Intel C/C$^{++}$ and the assembler implementations is "token threading". If you disable the support for packed opcodes (option `AMX_NO_PACKED_OPC`), the default switches to "direct threading", because it is faster. You can force "token threading" with the option `AMX_TOKENTHREADING`. Token threading has more decoding overhead (one memory access per P-code instruction), but direct threading requires patching opcodes in the P-code stream and token threading does not. Token threading is therefore a requirement when the compiled script runs from ROM, or when "packed opcodes" are in effect. The ANSI C version only supports "switch threading".

## • Summary of definitions

| | |
|---|---|
| `AMX_ANSIONLY` | Remove Unicode and UTF-8 support from the abstract machine. |
| `AMX_ALTCORE` | Use an alternative P-code interpreter (instead of the ANSI-C version). |
| `AMX_ASM` | Compile the assembler version of the P-code interpreter; implies `AMX_ALTCORE`. |
| `AMX_JIT` | Add support for the JIT run-time; implies `AMX_ALTCORE`. |
| `AMX_LIBPATH` | |
| | The name of the environment variable to use for locating dynamically loaded extension modules (see page 60). This environment variable only applies to Linux and UNIX. Its default value is "`AMXLIB`" —see page 6. |
| `AMX_NATIVE_CALL` | Calling convention of native functions (applies to `amx.c` and to extension modules). |
| `AMX_NATIVETABLE` | Add a native table with "fixed" functions, see page 63. |
| `AMX_NODYNALOAD` | Disable support for dynamically loadable extension modules, see the discussion at page 60. |
| `AMX_NO_MACRO_INSTR` | Disable support for supplemental & macro instructions — this flag is implied when the `AMX_JIT` flag is set, as macro instructions are not supported by the JIT run-times. |
| `AMX_NO_OVERLAY` | Disable support for the overlay system —this flag is implied when the `AMX_JIT` flag is set, as overlays are not supported by the JIT run-times. |
| `AMX_NO_PACKED_OPC` | Disable support for packed opcodes —this flag is implied when the `AMX_JIT` flag is set, as packed opcodes are not supported by the JIT run-times. |
| `AMX_NOPROPLIST` | Remove the get/set property functions from `AMXCORE.C`. |
| `AMX_NORANDOM` | Remove the pseudo-random number generator from `AMXCORE.C` —presumably for replacing it by a better pseudo-random number generator. |
| `AMX_TERMINAL` | For `AMXCONS.C`, do not use console functions (Win32 console, ANSI/VT100 or plain console). |
| `AMX_TOKENTHREADING` | Use "token threading" (only valid for the GNU C/C++, Intel C/C++ implementations, and the assembler implementations). Token threading is implied when packed opcodes are supported, and packed opcodes are supported by default. If you wish to use direct threading (which |

|  | is faster), you must disable packed opcodes (see option `AMX_NO_PACKED_OPC`). |
|---|---|
| `AMXAPI` | Calling convention of interface functions; this overrides any `CDECL` or `STDECL` macros. |
| `AMXEXPORT` | The "exported" attribute for initialization and clean-up functions of extension modules, when those extension modules are in a DLL. |
| `CDECL` | Sets `AMXAPI` to `__cdecl`, for compatibility with the assembler core. |
| `FIXEDPOINT` | For `AMXCONS.C`, add fixed point support, see also `FLOATPOINT` option. |
| `FLOATPOINT` | For `AMXCONS.C`, add floating point support, see the separate section below. |
| `NDEBUG` | Compile without assertions (all files). |
| `STDECL` | Sets `AMXAPI` to `__stdcall`, for compatibility with the assembler core. |
| `UNICODE, _UNICODE` | Enable Unicode in the console I/O module and possibly other auxiliary libraries. |
| `VT100` | For `AMXCONS.C`, use ANSI/VT100 terminal codes (implicit for Linux). |

All compiling examples (listed below) have as few command line options as needed. Consult the compiler documentation to add debugging information or to enable optimizations. The program that each of the examples compile is `pawnrun`, a simple P-code interpretor that is developed starting at .

As an aside, "project" and "workspace" files for Microsoft Visual C/C$^{++}$, (for the compiler and the Abstract Machine library source files) can be found in the "msvc" subdirectory of where PAWN is installed.

• ANSI C

**Borland C$^{++}$ version 3.1, 16-bit**

```
bcc pawnrun.c amx.c amxcore.c amxcons.c
```

*The 16-bit compiler in the Borland C$^{++}$ 5.0 package appears to have a few code generator errors, so either use an earlier version of the Borland compiler, or compile in 32-bit.*

**LCC-Win32, 32-bit**

```
lc pawnrun.c amx.c amxcons.c amxcore.c
```

## Microsoft Visual C/C++ version 5.0 or 6.0, 32-bit

```
cl pawnrun.c amx.c amxcons.c amxcore.c
```

*When running with warning level 4, option "-W4", Visual C/C++ issues a few warnings for unused function arguments.*

## Watcom C/C++ version 11.0, 32-bit

```
wcl386 pawnrun.c amx.c amxcore.c amxcons.c
```

The above list is far from comprehensive. The PAWN Abstract Machine is portable across many compilers and many operating systems/architectures.

### • Alternate P-code interpreter: GNU GCC

The PAWN abstract machine supports alternative, optimized, implementations for its core function: the P-code parser and interpreter. One such implementations is still written in C, but using non-standard extensions. The GNU GCC and the Intel C/C++ compilers support the required extensions.

The command lines in the next examples may wrap around, separated with "*(. . . )*" marks. These should be typed as a single line and without the "*(. . . )*" marks.

### GNU GCC for Linux, FreeBSD and OpenBSD

```
gcc -opawnrun -I../linux -DAMX_ALTCORE pawnrun.c amx.c amxcore.c (...)
(...) amxcons.c amxexec_gcc.c ../linux/getch.c -ldl
```

*For optimal performance, the P-code interpreter from `amx.c` is replaced: see the option `AMX_ALTCORE` and the new file `amxexec_gcc.c`. The console I/O functionality in `amxcons.c` relies on a function that reads keys in raw mode without echo; this is standard on DOS and Windows platforms, but must be implemented explicitly in Linux —`getch.c`. The abstract machine also supports dynamically loaded extension modules by default (see ). Dynamic linking requires the inclusion of the library `libdl`.*

### GNU GCC with direct threading (for Linux, FreeBSD and OpenBSD)

```
gcc -opawnrun -I../linux -DAMX_ALTCORE -DAMX_NO_PACKED_OPC pawnrun.c (...)
(...) amx.c amxcore.c amxcons.c amxexec_gcc.c ../linux/getch.c -ldl
```

*Even better performance is obtained with "direct threading", which the GNU GCC core also supports. Direct threading cannot use the full instruction set, however. The* `AMX_NO_PACKED_OPC` *macro disables support for packed opcodes.*

### GNU GCC plus ncurses (for Linux, FreeBSD and OpenBSD)

```
gcc -opawnrun -I../linux -DUSE_CURSES -DAMX_ALTCORE pawnrun.c amx.c (...)
```

```
(...) amxcore.c amxcons.c amxexec_gcc.c -ldl -lcurses
```

*This example uses the "ncurses" library for terminal support, instead of the standard (limited) vt100 terminal support. The ncurses library also replaces the special file* `getch.c`*.*

• Alternate P-code interpreter: assembler

Assembler implementations of the core of the Abstract Machine are currently available for the Intel Pentium platform (derivatives of an original version by Marc Peter), and for the ARM processors (architecture version 4 and higher).

For the Pentium platform, there are two "calling convention" issues in the assembler implementation (in addition to those mentioned at ):

◇ The convention with which `amx_exec_asm` itself is called. The default calling convention is Watcom's register calling convention. For other compilers, change this to `__cdecl` by setting the macro `STACKARGS`.

◇ The convention for calling the "hook" functions (the native function dispatcher and the debugger callback). Again, the default is Watcom's register calling convention. Use the macros `CDECL` or `STDECL` for `__cdecl` and `__stdcall` respectively. (Since `STDCALL` is a reserved word on the assembler, I had to choose a different name for the macro, hence `STDECL`.)

In `amx.c`, the calling convention for the hook functions is set with the `AMXAPI` macro. You may need to adjust the `AMXAPI` macro so that it does not conflict with the calling convention for the hook functions that the assembler core assumes.

Included in the archive are two pre-assembled object files, for those of you who do not have an assembler (note that several suitable assemblers are freely available, inclusing Microsoft's MASM). The two assembler files differ only in the calling convention used. Below are the filenames and the commands that I used to assemble them:

`amxexecc.obj` (`__cdecl` calling convention)

```
ml /c /DCDECL /DSTACKARGS /Cx /coff /Foamxexecc amxexec.asm
```

`amxexecs.obj` (_stdcall calling convention)

```
ml /c /DSTDECL /DSTACKARGS /Cx /coff /Foamxexecs amxexec.asm
```

The assembler file `amxexec.asm` is compatible with Microsoft MASM, Borland TASM and Watcom WASM. The Netwide assembler (NASM, a free assembler that runs on a variety of platforms) has a syntax that is similar to that of MASM/TASM/WASM, but is incompatible with it. The file "`amxexecn.asm`" (note the "`n`" after "`amxexec`") is the same implementation of the assembler core for the AMX, but using the "Netwide assembler" syntax. The Netwide assembler version of the AMX code does not support Watcom's "register calling" convention —it always uses the `__cdecl` for the `amx_exec_asm` function itself. The calling convention for the "hook" functions is `__cdecl` by default, but this can be changed to `__stdcall` by setting the STDECL macro at the NASM command line.

There are two versions for the ARM processor, matching two common assemblers for the platform: the assembler from ARM Ltd. itself ("`armasm`") and the GNU assembler. The files are `armexec_arm.s` and `armexec_gnuarm.s` respectively. The P-code interpreter is for architecture version 4 or higher (which includes the popular ARM7TDMI and Cortex-M3 processors) and compliant with the "ARM Architecture Procedure Call Standard (AAPCS)". By default, it builds for "ARM" mode, to build the P-code interpreter for "Thumb-2" mode, add the definition THUMB2 to the command line when building the assembler file.

### Borland C++ version 5.02 & TASM, 32-bit

```
bcc32 -DAMX_ASM -TdCDECL -TdSTACKARGS pawnrun.c amx.c amxcore.c (...)
(...) amxcons.c amxexec.asm
```

*You must assemble* `amxexec.asm` *with the "*`CDECL`*" and "*`STACKARGS`*" options. The "-T" compiler option passes what follows onto TASM32.*

### Borland C++ version 5.02 & NASM, 32-bit

```
nasmw -Ox -fobj -dBORLAND amxexecn.asm
bcc32 -DAMX_ASM pawnrun.c amx.c amxcore.c amxcons.c amxexecn.obj
```

*You must assemble* `amxexecn.asm` *with the "*`BORLAND`*" option, because Borland C++ uses different segment declarations as other compilers.*

### GNU GCC for Linux, FreeBSD and OpenBSD

```
nasm -Ox -f elf amxexecn.asm
gcc -o pawnrun -DAMX_ASM -I../linux pawnrun.c amx.c amxcore.c (...)
(...) amxcons.c ../linux/getch.c amxexecn.o -ldl
```

*Most Linux distributions use the "elf" file format. See for the*

*extra file* `getch.c` *and* *for the option* `-ldl` *which causes the inclusion of the library* `libdl`

## LCC-Win32 & MASM, 32-bit

```
ml /c /DCDECL /DSTACKARGS /Cx /coff amxexec.asm

lc -DAMX_ASM pawnrun.c amx.c amxcons.c amxcore.c amxexec.obj
```

*LCC-Win32 does not come with an assembler, I have used MASM here. I have only done preliminary testing with LCC-Win32.*

## Microsoft Visual C/C++ version 5.0 or 6.0, 32-bit, __cdecl

```
ml /c /DCDECL /DSTACKARGS /Cx /coff amxexec.asm

cl -Gd -DAMX_ASM pawnrun.c amx.c amxcons.c amxcore.c amxexec.obj
```

*Microsoft appears to use* `__cdecl` *calling convention by default, but I have forced the calling convention to be sure: option -Gd.*

## Microsoft Visual C/C++ version 5.0 or 6.0, 32-bit, __stdcall

```
ml /c /DSTDECL /DSTACKARGS /Cx /coff amxexec.asm

cl -Gz -DAMX_ASM -DAMXAPI=__stdcall pawnrun.c amx.c amxcons.c (...)

(...) amxcore.c amxexec.obj
```

*Option* `-Gz` *forces* `__stdcall` *calling convention. The assembler file now uses* `STDECL` *(for* `__stdcall`*) too.*

## Watcom C/C++ version 11.0 & WASM, 32-bit

```
wcl386 -l=nt -dAMX_ASM pawnrun.c amx.c amxcore.c amxcons.c amxexec.asm
```

*Watcom C/C++ uses register calling convention, which is fastest in this case.*

## GNU GCC for ARM (GNUARM, YAGARTO)

```
arm-elf-gcc -o pawnrun.elf -DAMX_ASM pawnrun.c amx.c amxcore.c (...)

(...) amxcons.c amxexec_gnuarm.s
```

*Many embedded systems lack a "console" for text input and output, so you will typically need to replace (or reimplement) the native functions in* `amxcons.c`*. This example line builds the abstract machine for ARM mode, to use Thumb-2 mode (e.g. for Cortex-M3), declare the* `THUMB2` *macro on the command line, in addition to* `AMX_ASM`*. Depending on the compiler configuration, you may also need to specify a CPU model or architecture.*

### • Just-In-Time compiler

The third option is to add the "Just-In-Time" run-time, plus support routines. The JIT translates the P-code of the AMX to native machine code at run-time. The resulting code is more than twice as fast as the assembler version of the Abstract Machine (which was pretty fast already). To add support for the JIT, you must define the macro "`AMX_JIT`" via a command line switch.

In addition to compiling with the `AMX_JIT` macro defined, the host application must:

◇ set `AMX_FLAG_JITC` in the "`flags`" field of the `AMX` structure *before* calling function `amx_Init`;

◇ call `amx_InitJIT` function *after* `amx_Init`.

The function `amx_InitJIT`, in turn, needs two extra memory blocks: one for the native machine instructions that the compiler generates and the other for any relocations. After `amx_InitJIT` returns, the relocation table buffer may be freed. The memory block holding the original PAWN P-code instructions is no longer needed and may also be freed.

Special care must be taken for the block that will contain the native machine code instructions: the permission to execute machine code from the memory block *must* be set for the block. On Intel processors, any block of memory that has "read access" implicitly has "execution access". To block the treat of buffer overruns that allow the execution of arbitrary code, AMD has introduced the "**n**o e**x**ecute" (NX) bit in the descriptor of a memory page, and Intel has adopted this design — though calling it "e**x**ecution **d**enied" (XD). On an operating system that has the NX/XD bit set by default, you must then make sure that the memory block into which the JIT run-time generates the instructions has the NX/XD bit *cleared*.

The JIT run-time itself needs only read-write access to the memory block for the native machine instructions (this is the default for a memory block that you allocate). The execution of the JIT-compiled code, through `amx_Exec`, requires full access to the memory block: read, write and execute. The block needs write access, because the `SYSREQ` opcode is patched to `SYSREQ.D` after the first lookup (this is an optimization, look up the address of the native function only once). On Microsoft Windows, function `VirtualAlloc` can allocate a block of memory with full access; alternatively `VirtualProtect` may change the access rights on an existing memory block. On versions of Linux that support the NX/XD bits, you can use `vmalloc_exec` to get a block with full access, or adjust the access rights on an already allocated block with function `mprotect`. If your version of

Linux does not provide `vmalloc_exec`, it will probably not support the NX/XD bit. For processors or operating systems that do not support the NX/XD bit, execution of code is implicitly allowed. You can use the standard `malloc` in place of `VirtualAlloc` and `vmalloc_exec`.

During compilation, the JIT run-time requires write-access to its own code segment: the JIT run-time patches P-code parameters into its own code segment during compilation. To make these patches possible, `amx_InitJIT` temporarily enables "write-access" it is own code segment, for operating systems that require this.

`amx_Init` gives a conservative estimate of the size of the memory block that is needed to compile the native machine code into. *Conservative estimate* means here that the memory block is guaranteed to be big enough, and will likely be far bigger than what is really needed. When `amx_InitJIT` returns, it has calculated the *real* required memory size. To save memory, you may therefore want to shrink or re-allocate the memory block after `amx_InitJIT` returns.

The toolkit comes with the source code of `prun_jit.c` which is a modification of the "PAWNRUN" program (the example program for the embedding of the abstract machine, see page 6) for the JIT run-time. This example program lays out the steps described above.

There are, in fact, three versions of the JIT, all of which are for the 80x86 processor architecture (Intel Pentium, AMD Athlon):

| | |
|---|---|
| `amxjitr.asm` | uses register based calling conventions and requires Watcom C/C++. It requires the Watcom assembler (WASM) to build. |
| `amxjits.asm` | uses `__cdecl` or `__stdcall` calling conventions (both are stack based) for other Windows' C/C++ compilers. It can be built with WASM or the portable derivative JWASM (both are open source). |
| `amxjitsn.asm` | is the same as `amxjits.asm`, but implemented in "NASM" and thereby making the JIT-compiler available to Linux and UNIX-like operating systems. |

Apart from the calling conventions and the assembler syntax, the three JIT versions are identical.

The source files `amxjitr.asm`, `amxjits.asm` and `amxjitsn.asm` contain several definitions with which you can trade performance for other options. See the source files for a description of these definitions.

## Borland C++ version 5.02, 32-bit

```
bcc32 -DAMX_JIT -Tm2 prun_jit.c amx.c amxcore.c amxcons.c amxjits.asm
```

*You must force TASM to use at two passes, so that forward references are resolved. The -Tm2 option accomplishes this.*

## Watcom C/C++ version 11.0, 32-bit

```
wcl386 -l=nt -dAMX_JIT prun_jit.c amx.c amxcore.c amxcons.c amxjitr.asm
```

*Watcom C/C++ uses register calling convention, which is fastest in this case.*

## GNU GCC for Linux, FreeBSD and OpenBSD

```
nasm -Ox -f elf amxjitsn.asm
gcc -o pawnrun -DAMX_JIT -I../linux prun_jit.c amx.c amxcore.c (...)
(...) amxcons.c ../linux/getch.c amxjitsn.o -ldl
```

*Most Linux distributions use the "elf" file format. The -O1 option adjusts the correct jump sizes (short/near) automatically. See for the extra file* `getch.c` *and for the option* -ldl *which causes the inclusion of the library* `libdl`.

The JIT does not support the debugger hook. That is, the JIT translates the P-code into native machine code for the processor, but it skips the BREAK opcode completely. When running the native machine code, the debugger hook function will therefore never be invoked. If you wish to have a solution where the script code can run at maximum speed, and at the same time be able to debug the script, one option is to include *both* the assembler core (see ) and the JIT in the abstract machine. In this case, the compiler command line would specify both the `AMX_JIT` and `AMX_ASM` macros, and both the assembler files for a JIT and an assembler core are added to the project.

With a combined JIT & assembler core, you can debug the script by running on the assembler core and run the retail code on the JIT. The host application can decide what "core" to run the script on before calling `amx_Init`.

## • Minimizing the size of the abstract machine

For embedded systems, the code space may be severely restricted and the size of the abstract machine must be minimized.

The first step in doing this is to add only the functions from the complete API that you actually need. This can be done by adding the macros related to the functions on the compiler command line.

| | |
|---|---|
| AMX_ALIGN | for `amx_Align16`, `amx_Align32` and `amx_Align64` |
| AMX_ALLOT | for `amx_Allot` and `amx_Release` |
| AMX_CLEANUP | for `amx_Cleanup` |
| AMX_CLONE | for `amx_Clone` |
| AMX_DEFCALLBACK | for `amx_Callback` |
| AMX_EXEC | for `amx_Exec` |
| AMX_FLAGS | for `amx_Flags` |
| AMX_INIT | for `amx_Init` and `amx_InitJIT` |
| AMX_MEMINFO | for `amx_MemInfo` |
| AMX_NAMELENGTH | for `amx_NameLength` |
| AMX_NATIVEINFO | for `amx_NativeInfo` |
| AMX_PUSHXXX | for `amx_Push`, `amx_PushAddress`, `amx_PushArray` and `amx_PushString` |
| AMX_RAISEERROR | for `amx_RaiseError` |
| AMX_REGISTER | for `amx_Register` |
| AMX_SETCALLBACK | for `amx_SetCallback` |
| AMX_SETDEBUGHOOK | for `amx_SetDebugHook` |
| AMX_UTF8XXX | for `amx_UTF8Get`, `amx_UTF8Put` and `amx_UTF8Check` |
| AMX_XXXNATIVES | for `amx_NumNatives`, `amx_GetNative` and `amx_FindNative` |
| AMX_XXXPUBLICS | for `amx_NumPublics`, `amx_GetPublic` and `amx_FindPublic` |
| AMX_XXXPUBVARS | for `amx_NumPubVars`, `amx_GetPubVar` and `amx_FindPubVar` |
| AMX_XXXSTRING | for `amx_StrLength`, `amx_GetString` and `amx_SetString` |
| AMX_XXXTAGS | for `amx_NumTags`, `amx_GetTag` and `amx_FindTagId` |
| AMX_XXXUSERDATA | for `amx_GetUserData` and `amx_SetUserData` |

When adding none of the above macros, *all* functions are included. When adding one or more, all functions (or function groups) that are *not* mentioned are removed from the code.

The second step is to make the abstract machine support a limited number of instructions. The complete abstract machine has 177 instructions, but an abstract machine needs only to support 74 —see also appendix D. By adding the macros `AMX_NO_OVERLAY`, `AMX_NO_MACRO_INSTR` and `AMX_NO_PACKED_OPC`, the groups for overlays, supplemental & macro instructions, and packed opcodes are disabled. The PAWN scripts for a minimal abstract machine need to be compiled with optimization level 1 and without overlay support (higher optimization levels use supplemental instructions).

## • Adding a terminal to the abstract machine

A simple text terminal is often convenient for users of a product, as it lets them print out text strings and get input in a plain and simple way. The strings printed on the console can also serve as a debugging or tracing aid for the user.

Example console functions are in the file AMXCONS.C, these allow for printing formatted text and reading keyboard input. The default implementation of the console interface writes to the standard output console for a "text mode" application: this is a "DOS box" for Microsoft Windows and the active terminal for Linux/UNIX. On Linux/UNIX, the functions support the VT100 terminal, and on Microsoft Windows the equivalent functionality is emulated. There is a fall-back using only the functions of standard C —this imposes several limitations, of course, but it works everywhere.

For better embedding in an application, you may want to write a custom terminal. As an example how to write the support code, the PAWN toolkit comes with two alternative terminal implementations:

termwin   A terminal for Microsoft Windows GUI ("windowed") applications. It may be compiled to use either ASCII/ANSI console I/O or Unicode. Although the number of columns and lines is fixed, the terminal window can be resized and scrolled, and the terminal allows the font to be scaled as well. This implementation supports multiple concurrent terminals.

term_ga   A terminal implemented in the cross-platform "GraphApp" library; it runs on Microsoft Windows, Linux, FreeBSD and the Macintosh. This terminal supports UTF-8 natively, and it may be compiled with Unicode ("wide character") support as well.

To compile with a special terminal, the default implementation of terminal I/O functions in AMXCONS.C must be disabled, and a source file with the desired terminal must be added to the project. With Watcom C/C$^{++}$ for the example, the command line for using termwin would be:

```
wcl386 -dAMX_TERMINAL -l=nt pawnrun.c amx.c amxcore.c amxcons.c termwin.c
```

If you compile the Microsoft Windows terminal for Unicode, you need to add the definition of the macro "UNICODE" on the command line. When you want the Unicode terminal to run as well in Microsoft Windows 9x, you will need to link against the "unicows" library (the "Microsoft Layer for Unicode" on Windows 95/98/ME). See the Microsoft site for details on Unicode and unicows.

Using the "GraphApp" terminal involves only slightly more work: GraphApp requires a redefinition of the entry point of the program (function `main`). The easiest way to get it running is to include the file "`grahpapp.h`" in `pawnrun.c`. Of course, the GraphApp libraries must be compiled as well.

## • Support for floating point in the Abstract Machine

The definitions for user defined operators for the floating point routines are in the file "`FLOAT.INC`". You can use floating point arithmetic in your PAWN programs by including this file. The include file gives definitions for native functions that perform the basic floating point operations and user-defined operators to map those to the common add/subtract/multiply/divide operators. See the PAWN booklet "The Language" for more information on user-defined operators.

The abstract machine needs to support floating point operations as well. This requires two or three additions to the compilation of the abstract machine:

1. you must define the macro "`FLOATPOINT`" when compiling the source files;
2. you should add the file `FLOAT.C` to the list of files
3. depending on the C compiler/linker, you may need to add a compiler option or a library file for the linker.

These two/three steps apply to all "compiler command lines" given above. For example, the first command line (ANSI C, using the 16-bit Borland C$^{++}$ compiler) becomes:

```
bcc -DFLOATPOINT pawnrun.c amx.c amxcore.c amxcons.c amxfloat.c
```

*The original line read:* `bcc pawnrun.c amx.c amxcore.c amxcons.c`

The Borland C$^{++}$ compiler requires no extra option to compile floating point programs. The GNU GCC compiler, however, must be instructed to add the "math" library to the linking phase, with the option `-lm`. The command line for GCC for Linux becomes:

```
gcc -o pawnrun -DFLOATPOINT -I../linux pawnrun.c amx.c (...)
```
`(...)` `amxcore.c amxcons.c amxfloat.c -lm`

Fixed point support, by the way, is added in nearly the same way: you add the macro `FIXEDPOINT` on the compiler command line and you include the file `FIXED.C` on the file list. In your PAWN program, you must include the file `FIXED.INC` for the definitions and user defined operators.

● Compiling "dynamically loadable" modules

The above section on adding floating point to the abstract machine did so by compiling/linking the support statically into the run-time. An alternative is to compile the abstract machine with only a minimal set of extension modules and native functions, and to create additional libraries as dynamically loadable modules (or "plug-ins").

To create a dynamically loadable extension module, the C/C++ file that implements the module must be built as a DLL (Microsoft Windows) or a shared library (UNIX/Linux).

In Microsoft Windows, the amx_*Filename*Init and amx_*Filename*Cleanup functions must be marked as "exported". For that purpose, `AMX.H` reserves the macro `AMXEXPORT`, which you use on the declaration of the functions amx_*Filename*Init and amx_*Filename*Cleanup, and then define it to the appropriate (compiler dependent) calling convention on the compiler command line. Exported functions in Microsoft Windows should also have the "`__stdcall`" calling convention, see below.

The exported function names should furthermore not be "mangled". In a C++ project the files should be declared `extern "C"` to avoid name mangling. Compilers for Microsoft Windows routinely mangle C functions as well (for example `amx_PowerInit` becomes `amx_PowerInit@4`), and this must then be explicitly disabled through a linker "`.DEF`" file or a compiler option. Watcom C/C++ uses an "`.LBC`" file instead of a `.DEF` file.

A complication in Microsoft Windows, next to name mangling, is the calling convention. It is common for Dynamic Link Libraries that the exported functions use the "`__stdcall`" calling convention. Technically, the *native* functions do not have to use the same calling convention as the exported functions (amx_*Filename*Init and amx_*Filename*Cleanup), but for reasons of similarity and interoperability, I advise that you also set the calling convention of native functions and of the "hook" functions to `__stdcall`. This, in turn, means that the abstract machine code must also be built with the `__stdcall` calling convention for native functions and hook functions (set both macros `AMX_NATIVE_CALL` and `AMXAPI` to `__stdcall`). Refer to page 72 for details.

An example command line to create the "floating point arithmetic" extension module as a DLL for Microsoft Windows, using Borland C++ 5.0 is:

```
bcc32 -tWD -eamxFloat -DAMXEXPORT="_export" (...)
```
```
(...) -DAMX_NATIVE_CALL=__stdcall -DAMXAPI=__stdcall amxfloat.c amx.c amxfloat.rc
```

Note that the host program should now also use the `__stdcall` calling convention for native functions and for the hook functions. The console I/O extension module (`AMXCONS.C`) also contains some support for fixed point and floating point values, which must be separately enabled —see the preceding section.

A native function library that is created as a DLL/shared library needs to link to a few functions in the file `amx.c` —notably `amx_Register`. It is, however, a waste of space to include all the functions in `amx.c` into the module: it is unlikely that the module will call `amx_Init` or `amx_Exec`, for example. To strip unneeded functionality from `amx.c`, define macros on the compiler command line to specify the set of functions that you *do* want:

| | |
|---|---|
| AMX_ALIGN | for `amx_Align16`, `amx_Align32` and `amx_Align64` |
| AMX_ALLOT | for `amx_Allot` and `amx_Release` |
| AMX_CLEANUP | for `amx_Cleanup` |
| AMX_CLONE | for `amx_Clone` |
| AMX_DEFCALLBACK | for `amx_Callback` |
| AMX_EXEC | for `amx_Exec` |
| AMX_FLAGS | for `amx_Flags` |
| AMX_INIT | for `amx_Init` and `amx_InitJIT` |
| AMX_MEMINFO | for `amx_MemInfo` |
| AMX_NAMELENGTH | for `amx_NameLength` |
| AMX_NATIVEINFO | for `amx_NativeInfo` |
| AMX_PUSHXXX | for `amx_Push`, `amx_PushAddress`, `amx_PushArray` and `amx_PushString` |
| AMX_RAISEERROR | for `amx_RaiseError` |
| AMX_REGISTER | for `amx_Register` |
| AMX_SETCALLBACK | for `amx_SetCallback` |
| AMX_SETDEBUGHOOK | for `amx_SetDebugHook` |
| AMX_UTF8XXX | for `amx_UTF8Get`, `amx_UTF8Put` and `amx_UTF8Check` |
| AMX_XXXNATIVES | for `amx_NumNatives`, `amx_GetNative` and `amx_FindNative` |
| AMX_XXXPUBLICS | for `amx_NumPublics`, `amx_GetPublic` and `amx_FindPublic` |
| AMX_XXXPUBVARS | for `amx_NumPubVars`, `amx_GetPubVar` and `amx_FindPubVar` |
| AMX_XXXSTRING | for `amx_StrLength`, `amx_GetString` and `amx_SetString` |
| AMX_XXXTAGS | for `amx_NumTags`, `amx_GetTag` and `amx_FindTagId` |
| AMX_XXXUSERDATA | for `amx_GetUserData` and `amx_SetUserData` |

# Using CMake

CMake is a cross-platform, open-source make system, which generates "`make-file`'s" or project files for diverse compilers and platforms. It runs natively on Linux, various UNIX variants and Windows (without requiring Cygwin) and supports various compilers. You can find more information on CMake plus a freely downloadable copy on http://www.cmake.org/.

The PAWN toolkit comes with two CMake project files. The first builds the compiler as a shared library and a console-mode "front end". The second builds a simple run-time program that embeds the abstract machine, a simple console debugger, and various extension modules as shared libraries. The CMake project files are in the "compiler" and "amx" subdirectories of where the PAWN toolkit source code is installed. When unpacking the PAWN source code from a `.ZIP` or `.TGZ` archive, the CMake project files are in these sub-directories below the directory where you unpacked the archive into.

## • Microsoft Windows

1. Launch `CMakeSetup`.

2. Select for the source code directory, the "compiler" or the "amx" sub-directory in the directory tree for the toolkit, depending of what you want to build.

   For example, if you installed the toolkit in `C:\Pawn`, the source directory for the compiler is `C:\Pawn\source\compiler`.

   You may also build both the compiler and the abstract machine in a single run by selecting the top-level source directory as the source directory of CMake.

3. Select as destination the "bin" sub-directory, or any other directory of your choice. The makefile (or project files) will be generated in the destination directory.

4. Select the compiler to use, as well. On Microsoft Windows, CMake supports Microsoft and Borland compilers, as well as GNU GCC.

5. Click on the "Configure" button. After an initial configuration, you may have items displayed in red. By this, CMake indicates that these items may need adjustment, but in the case of PAWN, this is rarely needed. Click "Configure" once more for the final configuration.

6. Click on the "OK" button. This exits the `CMakeSetup` program after creating a number of files in the destination subdirectory.

7. Build the program in the usual way. For Microsoft Visual C/C$^{++}$, CMake has created a Visual Studio project and "Workspace" files; for other compilers CMake builds a `makefile`.

● Linux / UNIX

1. Change to the directory where you want to generate the makefile (and build the executable). Typically, this will be the "bin" sub-directory in the directory tree for the toolkit. For example, if you installed the toolkit in `/opt/Pawn`, the bin sub-directory for the compiler is `/opt/Pawn/bin`.

   If you installed PAWN as "root", then you also need to be root to recompile PAWN.

2. Launch "`ccmake ../source/compiler`" or "`ccmake ../source/amx`", depending on what you want to build (the compiler or the abstract machine). You can build both at the same time with "`ccmake ../source`".

3. Press the "c" key for "configure". After an initial configuration, you may have items in the list that have a "*" in front of their value. By this, CMake indicates that these items may need adjustment, but in the case of PAWN, this is rarely needed. Type "c" once more for the final configuration.

4. Press the "g" button for "generate and quit". Then build the program by typing "`make`".

5. Optionally, you can then also do a "`make install`" to copy the executable files to a path chosen with `ccmake`.

# Abstract Machine design and reference

The first issue is: why an abstract machine at all? By compiling into the native machine language of the processor of your choice, the performance will be so much better.

A few reasons are:

◇ Cross-platform compatibility of the compiled binary code. PAWN scripts now run on a variety of processors and microcontrollers (8-bit to 64-bit), with and without operating system. of different architectures.

◇ It is far easier to keep a program running in an abstract machine inside its "sandbox". For example, an unbounded recursion in an abstract machine crashes the abstract machine itself, but not much else. If you run native machine code, the recursive routine may damage the system stack and crash the application. Although modern operating systems support multi-threading, with a separate stack per thread, the default action for an overrun of any stack is still to shut down the entire application.

◇ It is easier to design a language where a data object (an array) can contain P-code which is later executed. Modern operating systems separate code and data sections: you cannot write into a code section and you cannot execute data; that is, not without serious effort.

The current PAWN language does not have the ability to execute P-code from an array, but the abstract machine is not too tightly coupled to the language. That is, future versions of the PAWN language may provide a means to execute a code stream from a variable without requiring me to redesign the abstract machine.

My first stab at designing an abstract machine was to look at current implementations. It appears that it is some kind of a tradition to implement abstract machines as stack machines, even though the design for microprocessors has moved towards register based implementations. All the abstract machines I encountered are stack based. These include:

◇ the B language (predecessor of C)
◇ BOB
◇ Euphoria
◇ Java VM (JVM)

◇ Lua (before version 5)
◇ Microsoft C/C$^{++}$ 7.0 (P-code option)
◇ the Amsterdam Compiler Kit
◇ QuakeC VM

Stack machines are surely compact, flexible and simple to implement, but they are also more difficult to optimize for speed. To see why, let's analyze a specific example.

```
a = b + 2;      /* where "a" and "b" are simple variables */
```

**Native code**
In 32-bit assembler, this would be:

```
mov     eax, [b]
add     eax, 2
mov     [a], eax
```

**Stack based abstract machine**
Forth is the archetype for a stack machine, I will therefore use it as an example. The same routine in Forth would be:

```
b @ 2 + a !
```

where each letter is an instruction (the "@" stands for "fetch" and "!" for store; note that stack machines run code in "reverse polish notation"). So these are six instructions in P-code, but the code expands to:

```
b       push    offset b
@       pop     eax
        push    [eax]
2       push    2
+       pop     edx
        pop     eax
        add     eax, edx
        push    eax
a       push    offset a
!       pop     edx
        pop     eax
        mov     [edx], eax
```

Two observations: **1**. the stack machine makes heavy use of memory (bad for performance) and **2**. the expanded code is quite large when compared to the native code (12 instructions versus 3).

The expanded code is what a "just-in-time" compiler (JIT) might make from it (though one may expect an optimizing JIT to reduce the redundant "pushes" and "pops" somewhat). When running the code in an abstract machine, the abstract machine must also expand the code, but in addition, it has overhead for fetching and decoding instructions. This overhead is at least two native instructions per P-code instruction (more on this later). For six P-code instructions, one should add another 12 native instructions to the 12 native instructions of the expanded

code. And still, the example is greatly simplified, because the code runs on the systems stack and uses the systems address space.

In other words, a stack-based abstract machine runs a native 3-instruction code snippet in 6 P-code instructions, which turn out to take 24 native instructions, and more if you want to run the abstract machine on its own stack and in its own (protected) data space.

### Register-based abstract machine

Microprocessors have used registers since their theoretical inception by Von Neumann. Extending this architecture to an abstract machine is only natural. There are two advantages: the abstract machine instructions map better to the native instructions (you may actually use the processor's registers to implement the abstract machine's registers) and the number of virtual instructions that is needed to executed a simple expression can be reduced.

As an example, here is the code for the PAWN "AMX", a two-register abstract machine (AMX stands for "Abstract Machine eXecutor"):

```
load.pri    b   ; "pri" is the primary register, i.e. the accumulator
const.alt   2   ; "alt" is the alternate register
add             ; pri = pri + alt
stor.pri    a   ; store "pri" in variable "a"
```

In expanded code, this would be:

```
load.pri   b        mov   eax, [b]
const.alt  2        mov   edx, 2
add                 add   eax, edx
stor.pri   a        mov   [a], eax
```

The four bytecode instructions map nicely to native instructions. Here again, we will have to add the overhead for fetching and decoding the P-code instructions (2 native instructions per P-code instruction). When compared to a stack-based abstract machine, the register-based abstract machine runs twice as fast; in 12 native instructions, versus 24 native instructions for a stack-based abstract machine.

There is more: in my experience, stack-based abstract machines are easier to optimize for size and register-based abstract machines are easier to optimize for speed. So a register-based abstract machine can indeed be twice as fast as a stack-based abstract machine.

To elaborate a little further on optimizing: I have intentionally chosen to add "2" to a variable. Incrementing or decrementing a value by one or two is such a common case that Forth has a special operator for them: the word "2+" adds 2 to a value. Assuming that a good (stack-based) abstract machine also has special

opcodes for common operations, using this "2+" word instead of the general words "2" and "+" removes one P-code instruction and 3 native instructions. This would bring the native instruction count down to 21. However, the same optimization trick applies to the register-based abstract machine. The PAWN abstract machine has an "add.c" opcode that adds a constant value to the primary register. The optimized sequence would be:

```
load.pri  b        mov   eax, [b]
add.c     2        add   eax, 2
stor.pri  a        mov   [a], eax
```

which results to 3 native instructions plus 6 instructions of overhead for fetching and decoding the P-code instructions. The register-based abstract machine (which needs 9 native instructions) is still approximately twice as fast as the stack-based abstract machine (at 21 native instructions).

## • Threading

In a "token threaded" interpreter, each opcode is an index in a table that contains a "jump address" for every instruction. In a "direct threaded" interpreter, the opcode *is* the jump address itself. Direct threading often requires that all opcodes are *relocated* to jump addresses upon compilation or upon loading a pre-compiled file. The file format of the PAWN abstract machine is designed such that both token threading and direct threading are possible.

A threaded abstract machine is conventionally written in assembler, because most high level languages cannot store label addresses in an array. The GNU C compiler (GCC), however, extends the C language with an unary "&&" operator that returns the address of a label. This address can be stored in a "`void *`" variable type and it can be used later in a `goto` instruction. Basically, the following snippet does the same a "`goto home`":

```
void *ptr = &&home;
goto *ptr;
```

The ANSI C version of the abstract machine uses a large `switch` statement to choose the correct instructions for every opcode. Due to direct threading, the GNU C version of the abstract machine runs approximately twice as fast as the ANSI C version. Fortunately, GNU C runs on quite a few platforms. This means that the fast GNU C version is still fairly portable.

## • Optimizing in assembler

The following discussion assumes an Intel 80386 or compatible processor. The same technique also applies to 16-bit processors and to processors of other brands, but the names (and number) of registers will be different.

It is beneficial to use the processor's registers to implement the registers of the abstract machine. The details of the abstract machine for the PAWN system follow later on in this appendix. Further assumptions are:

◇ PRI is an alias for the processor's register EAX and ALT is EDX

◇ ESI is the code instruction pointer (CIP)

◇ EDI points to the start of the data segment, ECX is the stack pointer (STK), EBX is the frame pointer (FRM) and EBP is available as a general purpose intermediate register; the remaining registers in the AMX (STP and HEA) are local variables.

Every opcode has a set of machine instructions attached to it, plus a trailer that branches to the next instruction. The trailer is identical for every opcode. As an example, below is the implementation of the ADD.C opcode:

```
add     eax, [esi]      ; add constant
add     esi, 4          ; skip constant
; the code below is the same for every instruction
add     esi, 4          ; pre-adjust instruction pointer
jmp     [esi-4]         ; jump to address
```

Note that the "trailer" which chains to the next instruction via (direct) threading consists of two instructions; this trailer was the origin of the premise of a 2-instruction overhead for instruction fetching and decoding in the earlier analysis.

In the implementation of the abstract machine, one can hand-optimize the sequences further. In the above example, the two "add esi, 4" instructions can, of course, be folded into a single instruction that adds 8 to ESI.

## Abstract Machine reference

The abstract machine consists of a set of registers, a proposed (or imposed) memory layout and a set of instructions. Each is discussed in a separate section.

## • Register layout

The abstract machine mimics a dual-register processor. In addition to the two "general purpose" registers, it has a few internal registers. Below is the list with the names and description of all registers:

| | |
|---|---|
| `PRI` | primary register (ALU, general purpose). |
| `ALT` | alternate register (general purpose). |
| `FRM` | stack frame pointer, stack-relative memory reads and writes are relative to the address in this register. |
| `CIP` | code instruction pointer. |
| `DAT` | offset to the start of the data. |
| `COD` | offset to the start of the code. |
| `STP` | stack top. |
| `STK` | stack index, indicates the current position in the stack. The stack runs downwards from the `STP` register towards zero. |
| `HEA` | heap pointer. Dynamically allocated memory comes from the heap and the `HEA` register indicates the top of the heap. |

Notably missing from the register set is a "flags" register. The abstract machine keeps no separate set of flags; instead all conditional branches are taken depending on the contents of the PRI register.

### • Memory image

The heap and the stack share a memory block. The stack grows downwards from STP towards zero; the heap grows upwards. An exception occurs when the STK and the HEA registers collide. (An exception means that the abstract machine aborts with an error message. There is currently no exception trapping mechanism.)

Figure 1 is a proposed memory image layout, and one that the standard Abstract Machine assumes for a self-contained AMX "job". Alternative layouts are possible. For instance, when you "clone" an AMX job, the new job will share the Prefix and the Code sections with the original job, and have the Data/Heap/Stack sections in a different memory block. Specifically, an implementation may choose to keep the heap and the stack in a separate memory block next to the memory block for the code, the data and the prefix. The top of the figure represents the lowest address in memory.

The binary file (on disk) consists of the "prefix", and the code and data sections. The heap and stack sections are not stored in the binary file, the abstract machine can build them from information in the "prefix" section. The prefix also contains start-up information, and the definitions of native and public functions.

Symbolic (debug) information may follow the code and data sections in the file. This symbolic information is typically not read into memory (at least not by the abstract machine). See appendix E for details.

FIGURE 1: *Memory layout of the abstract machine*

All multi-byte values in the prefix are stored with the low byte at the lower address (Little Endian, or "low byte first"). The byte order in the generated code and data sections is also in Little Endian.

| size | 4 bytes | size of the memory image, excluding the stack/heap |
|---|---|---|
| magic | 2 bytes | indicates the format and cell size |
| file_version | 1 byte | file format version, currently 8 |
| amx_version | 1 byte | required minimal version of the abstract machine |
| flags | 2 bytes | flags, see below |
| defsize | 2 bytes | size of a structure in the "native functions" and the "public functions" tables |
| cod | 4 bytes | offset to the start of the code section |
| dat | 4 bytes | offset to the start of the data section |
| hea | 4 bytes | initial value of the heap, end of the data section |
| stp | 4 bytes | stack top value (the total memory requirements) |
| cip | 4 bytes | starting address (main() function), -1 if none |
| publics | 4 bytes | offset to the "public functions" table |
| natives | 4 bytes | offset to the "native functions" table |
| libraries | 4 bytes | offset to the table of libraries |
| pubvars | 4 bytes | offset to the "public variables" table |
| tags | 4 bytes | offset to the "public tags" table |
| nametable | 4 bytes | offset to the symbol name table |
| overlays | 4 bytes | offset to the overlay table |

| *publics table* | variable | public functions table (see below) |
|---|---|---|
| *natives table* | variable | native functions table (see below) |
| *library table* | variable | library table (see below) |
| *pubvars table* | variable | public variables table (see below) |
| *tags table* | variable | public tags table (see below) |
| *overlay table* | variable | the overlay table (see below) |
| *name table* | variable | the symbol name table (see below) |

The `magic` value indicates the size of a cell in the P-code of the compiled program. This value is (in hexadecimal):

`F1E0`  for a 32-bit cell;

`F1E1`  for a 64-bit cell;

`F1E2`  for a 16-bit cell.

Each bit in the `flags` field contains one setting. Currently, the defined bits are (bits that are not mentioned are currently not defined):

    0  `AMX_FLAG_OVERLAY` —if set, the file is built with overlays

    1  `AMX_FLAG_DEBUG` —if set, the file contains symbolic (debug) information

    2  `AMX_FLAG_NOCHECKS` —if set, opcodes are a single byte (instead of cell-sized)

    3  `AMX_FLAG_SLEEP` —if set, the script uses the "`sleep`" instruction, which may cause the script to be re-entrant

    5  `AMX_FLAG_DSEG_INIT` —if set, the host program already initialized the data section of the abstract machine; a host application usually needs to do this (and set this flag) to run scripts from ROM.

   11 *reserved* —this bit is used internally

   12 *reserved* —this bit is used internally

   13 *reserved* —this bit is used internally

   14 *reserved* —this bit is used internally

   15 *reserved* —this bit is used internally

The fixed part of the prefix followed by a series of tables. Each table contains zero or more records. The name table has a variable record size; the size of the records in the other tables is in the `defsize` field in the prefix. To find the number of records in a table, subtract the offset to the table from the offset to the successive table, and divide that by `defsize`. For example, the number of records in the *natives table* is:

$$records = \frac{\texttt{libraries} - \texttt{natives}}{\texttt{defsize}}$$

The P-code follows the prefix immediately, but note that the prefix may be padded in order to align the code and data sections (this is a compiler option). The `cod` field in the header is the file offset to the start of the P-code.

Early versions of the P-code file format stored the symbol names directly in the records (which limited the maximum name length). This manual only covers the recent versions of the file format. These recent file formats include a "name table", which holds the symbol names for the symbols that the other tables refer to. Each name is in a variable sized record as a zero-terminated string. The advantage of this schema is that it allows for arbitrarily long symbol names while storing these names in a compact fashion.

The "public functions" table consists of a series of records of the following format:

| address | 4 bytes | the address (relative to COD) of the function |
|---------|---------|-----------------------------------------------|
| nameofs | 4 bytes | offset to the symbol name, relative to prefix |

The second field in the record is a byte offset to the function name, relative to the start of the P-code file —this offset points into the name table. The function address is relative to the "code section" in the file (the `COD` register holds the offset of the code segment relative to the start of the P-code file).

The native function table has the same lay-out as the public function table, with the exception that the function "`address`" is used internally, and should be zero in the P-code file. The library table has the same format as the native functions table. The "`address`" field is used internally and should be zero in the file.

The "public variables" table, again, has a similar record lay out as the public functions table. The address field of a public variable contains the variable's address relative to the DAT section.

The "tags" table uses the same format as well. This table only holds tags whose name or number might be useful to the host application or extension modules: tags that are used with the `exit` or `sleep` instructions or used with the `tagof` operator. The "`address`" field of a tag record contains the tag identifier.

The overlay table holds the file offset and the size of each overlay. In the current implementation, if overlays are active, every function is a separate overlay.

| offset | 4 bytes | offset of the start of the overlay |
|--------|---------|------------------------------------|
| size   | 4 bytes | size of the overlay, in bytes      |

• Instruction reference

Every instruction consists of an opcode followed by zero or one parameters. Each opcode is one byte in size; an instruction parameter has the size of a cell (usually four bytes). A few "debugging" instructions (at the end of the list) form an exception to these rules: they have two or more parameters and those parameters are not always cell sized.

Many instructions have implied registers as operands. This reduces the number of operands that are needed to decode an instruction and, hence, it reduces the time needed to decode an instruction. In several cases, the implied register is part of the name of the opcode. For example, `PUSH.pri` is the name of the opcode that stores the PRI register on the stack. This instruction has no parameters: its parameter (PRI) is implied in the opcode name.

The instruction reference is ordered by opcode. The description of two opcodes is sometimes combined in one row in the table, because the opcodes differ only in a source or a destination register. In these cases, the opcodes and the variants of the registers are separated by a "/".

The "semantics" column gives a brief description of what the opcode does. It uses the C language syntax for operators, which are the same as those of the PAWN language. An item between square brackets indicates a memory access (relative to the DAT register, except for jump and call instructions). So, `PRI = [address]` means that the value read from memory at location `DAT + address` is stored in PRI.

| core instructions | | | |
|---|---|---|---|
| # | **mnemonic** | **operand** | **semantics** |
| 0 | NOP | | no-operation, for code alignment |
| 1/2 | LOAD.pri/alt | address | PRI/ALT = [address] |
| 3/4 | LOAD.S.pri/alt | offset | PRI/ALT = [FRM + offset] |
| 5/6 | LREF.S.pri/alt | offset | PRI/ALT = [ [FRM + offset] ] |
| 7 | LOAD.I | | PRI = [PRI] (full cell) |
| 8 | LODB.I | number | PRI = "number" bytes from [PRI] (read 1/2/4 bytes) |
| 9/10 | CONST.pri/alt | value | PRI/ALT = value |
| 11/12 | ADDR.pri/alt | offset | PRI/ALT = FRM + offset |
| 13 | STOR | address | [address] = PRI |
| 14 | STOR.S | offset | [FRM + offset] = PRI |
| 15 | SREF.S | offset | [ [FRM + offset] ] = PRI |
| 16 | STOR.I | | [ALT] = PRI (full cell) |
| 17 | STRB.I | number | "number" bytes at [ALT] = PRI (write 1/2/4 bytes) |

| 18 | ALIGN.pri | number | Little Endian: PRI ^= *cell size*− number |
|---|---|---|---|
| 19 | LCTRL | index | PRI is set to the current value of any of the special registers. The index parameter must be: 0=COD, 1=DAT, 2=HEA, 3=STP, 4=STK, 5=FRM, 6=CIP (of the next instruction) |
| 20 | SCTRL | index | set the indexed special registers to the value in PRI. The index parameter must be: 2=HEA, 4=STK, 5=FRM, 6=CIP |
| 21 | XCHG | | Exchange PRI and ALT |
| 22/23 | PUSH.pri/alt | | [STK] = PRI/ALT, STK = STK − *cell size* |
| 24 | PUSHR.pri | | [STK] = PRI + DAT, STK = STK − *cell size* |
| 25/26 | POP.pri/alt | | STK = STK + *cell size*, PRI/ALT = [STK] |
| 27 | PICK | offset | PRI = [STK + offset] |
| 28 | STACK | value | ALT = STK, STK = STK + value |
| 29 | HEAP | value | ALT = HEA, HEA = HEA + value |
| 30 | PROC | | [STK] = FRM, STK = STK − *cell size*, FRM = STK |
| 31 | RET | | STK = STK + *cell size*, FRM = [STK], STK = STK + *cell size*, CIP = [STK], The RET instruction cleans up the stack frame and returns from the function to the instruction after the call. |
| 32 | RETN | | STK = STK + *cell size*, FRM = [STK], STK = STK + *cell size*, CIP = [STK], STK = STK + [STK] The RETN instruction removes a specified number of bytes from the stack. The value to adjust STK with must be pushed prior to the call. |
| 33 | CALL | offset | [STK] = CIP + 2x*cell size*, STK = STK − *cell size*, CIP = CIP + offset The CALL instruction jumps to an address after storing the address of the next sequential instruction on the stack. The address jumped to is relative to the current CIP, but the address on the stack is an absolute address. |
| 34 | JUMP | offset | CIP = CIP + offset (jump to the address relative from the current position) |
| 35 | JZER | offset | if PRI == 0 then CIP = CIP + offset |
| 36 | JNZ | offset | if PRI != 0 then CIP = CIP + offset |
| 37 | SHL | | PRI = PRI $\ll$ ALT |
| 38 | SHR | | PRI = PRI $\gg$ ALT (without sign extension) |
| 39 | SSHR | | PRI = PRI $\gg$ ALT with sign extension |

| 40/41 | SHL.C.pri/alt | value | PRI/ALT = PRI/ALT $\ll$ value |
|---|---|---|---|
| 42 | SMUL | | PRI = ALT * PRI (signed multiply) |
| 43 | SDIV | | PRI = ALT / PRI (signed divide), ALT = ALT mod PRI |
| 44 | ADD | | PRI = ALT + PRI |
| 45 | SUB | | PRI = ALT $-$ PRI |
| 46 | AND | | PRI = ALT & PRI |
| 47 | OR | | PRI = ALT $\vert$ PRI |
| 48 | XOR | | PRI = ALT ^ PRI |
| 49 | NOT | | PRI = !PRI |
| 50 | NEG | | PRI = $-$PRI |
| 51 | INVERT | | PRI = ~PRI |
| 52 | EQ | | PRI = PRI == ALT ? 1 : 0 |
| 53 | NEQ | | PRI = PRI != ALT ? 1 : 0 |
| 54 | SLESS | | PRI = PRI $<$ ALT ? 1 : 0 (signed) |
| 55 | SLEQ | | PRI = PRI $\leq$ ALT ? 1 : 0 (signed) |
| 56 | SGRTR | | PRI = PRI $>$ ALT ? 1 : 0 (signed) |
| 57 | SGEQ | | PRI = PRI $\geq$ ALT ? 1 : 0 (signed) |
| 58/59 | INC.pri/alt | | PRI = PRI + 1 / ALT = ALT + 1 |
| 60 | INC.I | | [PRI] = [PRI] + 1 |
| 61/62 | DEC.pri/alt | | PRI = PRI $-$ 1 / ALT = ALT $-$ 1 |
| 63 | DEC.I | | [PRI] = [PRI] $-$ 1 |
| 64 | MOVS | number | Copy memory from [PRI] to [ALT]. The parameter specifies the number of bytes. The blocks should not overlap. |
| 65 | CMPS | number | Compare memory blocks at [PRI] and [ALT]. The parameter specifies the number of bytes. The blocks should not overlap. PRI is zero if the blocks are equal. |
| 66 | FILL | number | Fill memory at [ALT] with value in [PRI]. The parameter specifies the number of bytes, which must be a multiple of the cell size. |
| 67 | HALT | 0 | Abort execution (exit value in PRI), parameters other than 0 have a special meaning. |
| 68 | BOUNDS | value | Abort execution if PRI $>$ value or if PRI $<$ 0 |
| 69 | SYSREQ | value | call system service |
| 70 | SWITCH | offset | Compare PRI to the values in the case table (whose address is passed as an offset from CIP) and jump to the address in the matching record. |
| 71/72 | SWAP.pri/alt | | [STK] = PRI/ALT and PRI/ALT = [STK] |

| 73 | BREAK | | conditional breakpoint —see appendix E |
|----|-------|--|----------------------------------------|
| 74 | CASETBL | . . . | A variable number of case records follows this opcode, where each record takes two cells. See the notes below for details on the case table lay-out. |

**reserved instructions (patched at run-time)**

| # | mnemonic | operand | semantics |
|---|----------|---------|-----------|
| 75 | SYSREQ.D | | variant of SYSREQ to call system service directly |
| 76 | SYSREQ.ND | | variant of SYSREQ.N to call system service directly |

**overlay instructions**

| # | mnemonic | operand | semantics |
|---|----------|---------|-----------|
| 77 | CALL.OVL | ovl | [STK] = (CIP + 2x*cell size*) & *overlay index*, STK = STK − *cell size*, *overlay index* = ovl The ICALL instruction branches to another overlay after storing current overlay index and the offset to the next sequential instruction on the stack. |
| 78 | RETN.OVL | | STK = STK + *cell size*, FRM = [STK], STK = STK + *cell size*, CIP/*overlay index* = [STK], STK = STK + [STK] The IRETN instruction removes a specified number of bytes from the stack. The value to adjust STK with must be pushed prior to the call. |
| 79 | SWITCH.OVL | offset | Compare PRI to the values in the case table (whose address is passed as an offset from CIP) and branch to the overlay in the matching record. |
| 80 | CASETBL.OVL | . . . | A variable number of case records follows this opcode, where each record takes two cells. See the notes below for details on the case table lay-out. |

**supplemental & macro instructions**

| # | mnemonic | operand | semantics |
|---|----------|---------|-----------|
| 81 | LIDX | | PRI = [ ALT + (PRI × *cell size*) ] |
| 82 | LIDX.B | shift | PRI = [ ALT + (PRI << *shift*) ] |
| 83 | IDXADDR | | PRI = ALT + (PRI × *cell size*) (calculate indexed address) |
| 84 | IDXADDR.B | shift | PRI = ALT + (PRI << *shift*) (calculate indexed address) |
| 85 | PUSH.C | value | [STK] = value, STK = STK − *cell size* |
| 86 | PUSH | address | [STK] = [address], STK = STK − *cell size* |
| 87 | PUSH.S | offset | [STK] = [FRM + offset], STK = STK − *cell size* |
| 88 | PUSH.ADR | offset | [STK] = FRM + offset, STK = STK − *cell size* |
| 89 | PUSHR.C | value | [STK] = value + DAT, STK = STK − *cell size* |

| 90 | PUSHR.S | offset | [STK] = [FRM + offset] + DAT, STK = STK − *cell size* |
| 91 | PUSHR.ADR | offset | [STK] = FRM + offset + DAT, STK = STK − *cell size* |
| 92 | JEQ | offset | if PRI == ALT then CIP = CIP + offset |
| 93 | JNEQ | offset | if PRI != ALT then CIP = CIP + offset |
| 94 | JSLESS | offset | if PRI < ALT then CIP = CIP + offset (signed) |
| 95 | JSLEQ | offset | if PRI <= ALT then CIP = CIP + offset (signed) |
| 96 | JSGRTR | offset | if PRI > ALT then CIP = CIP + offset (signed) |
| 97 | JSGEQ | offset | if PRI >= ALT then CIP = CIP + offset (signed) |
| 98 | SDIV.INV | | PRI = PRI / ALT (signed divide), ALT = PRI mod ALT |
| 99 | SUB.INV | | PRI = PRI − ALT |
| 100 | ADD.C | value | PRI = PRI + value |
| 101 | SMUL.C | value | PRI = PRI * value |
| 102/103 | ZERO.pri/alt | | PRI/ALT = 0 |
| 104 | ZERO | address | [address] = 0 |
| 105 | ZERO.S | offset | [FRM + offset] = 0 |
| 106 | EQ.C.pri | value | PRI = PRI == value ? 1 : 0 |
| 107 | EQ.C.alt | value | PRI = ALT == value ? 1 : 0 |
| 108 | INC | address | [address] = [address] + 1 |
| 109 | INC.S | offset | [FRM + offset] = [FRM + offset] + 1 |
| 110 | DEC | address | [address] = [address] − 1 |
| 111 | DEC.S | offset | [FRM + offset] = [FRM + offset] − 1 |
| 112 | SYSREQ.N | addr n | *macro*: PUSH.C *n*; SYSREQ.C *adr*; STACK *n+4* |
| 113 | PUSHM.C | n c... | replaces *n* PUSH.C opcodes |
| 114 | PUSHM | n a... | replaces *n* PUSH opcodes |
| 115 | PUSHM.S | n o... | replaces *n* PUSH.S opcodes |
| 116 | PUSHM.ADR | n o... | replaces *n* PUSH.ADR opcodes |
| 117 | PUSHRM.C | n c... | replaces *n* PUSHR.C opcodes |
| 118 | PUSHRM.S | n o... | replaces *n* PUSHR.S opcodes |
| 119 | PUSHRM.ADR | n o... | replaces *n* PUSHR.ADR opcodes |
| 120 | LOAD2 | a1 a2 | PRI = [a1], ALT = [a2] |
| 121 | LOAD2.S | o1 o2 | PRI = [FRM + o1], ALT = [FRM + o2] |
| 122 | CONST | adr val | [adr] = val |
| 123 | CONST.S | off val | [FRM + off] = val |

**packed instructions**

| # | **mnemonic** | **operand** | **semantics** |
| --- | --- | --- | --- |
| 124/125 | LOAD.P.pri/alt | | see LOAD.pri/alt |
| 126/127 | LOAD.P.S.pri/alt | | see LOAD.S.pri/alt |

| 128/129 | LREF.P.S.pri/alt | | see LREF.S.pri/alt |
|---|---|---|---|
| 130 | LODB.P.I | | see LODB.I |
| 131/132 | CONST.P.pri/alt | | see CONST.pri/alt |
| 133/134 | ADDR.P.pri/alt | | see ADDR.pri/alt |
| 135 | STOR.P | | see STOR |
| 136 | STOR.P.S | | see STOR.S |
| 137 | SREF.P.S | | see SREF.S |
| 138 | STRB.P.I | | see STRB.I |
| 139 | LIDX.P.B | | see LIDX.B |
| 140 | IDXADDR.P.B | | see IDXADDR.B |
| 141 | ALIGN.P.pri | | see ALIGN.pri |
| 142 | PUSH.P.C | | see PUSH.C |
| 143 | PUSH.P | | see PUSH |
| 144 | PUSH.P.S | | see PUSH.S |
| 145 | PUSH.P.ADR | | see PUSH.ADR |
| 146 | PUSHR.P.C | | see PUSHR.C |
| 147 | PUSHR.P.S | | see PUSHR.S |
| 148 | PUSHR.P.ADR | | see PUSHR.ADR |
| 149 | PUSHM.P.C | c. . . | see PUSHM.C |
| 150 | PUSHM.P | a. . . | see PUSHM |
| 151 | PUSHM.P.S | o. . . | see PUSHM.S |
| 152 | PUSHM.P.ADR | o. . . | see PUSHM.ADR |
| 153 | PUSHRM.P.C | c. . . | see PUSHRM.C |
| 154 | PUSHRM.P.S | o. . . | see PUSHRM.S |
| 155 | PUSHRM.P.ADR | o. . . | see PUSHRM.ADR |
| 156 | STACK.P | | see STACK |
| 157 | HEAP.P | | see HEAP |
| 158/159 | SHL.P.C.pri/alt | | see SHL.C.pri/alt |
| 160 | ADD.P.C | | see ADD.C |
| 161 | SMUL.P.C | | see SMUL.C |
| 162 | ZERO.P | | see ZERO |
| 163 | ZERO.P.S | | see ZERO.S |
| 164/165 | EQ.P.C.pri/alt | | see EQ.C.pri/alt |
| 166 | INC.P | | see INC |
| 167 | INC.P.S | | see INC.S |
| 168 | DEC.P | | see DEC |
| 169 | DEC.P.S | | see DEC.S |

| 170 | MOVS.P |  | see MOVS |
|-----|--------|--|----------|
| 171 | CMPS.P |  | see CMPS |
| 172 | FILL.P |  | see FILL |
| 173 | HALT.P |  | see HALT |
| 174 | BOUNDS.P |  | see BOUNDS |

● Branching

With a few exceptions, branch instructions (CALL, JUMP, etc.) use relative target addresses. The parameters of these opcodes are offsets relative to the address of the opcode itself. The offset can be both positive and negative. Using relative branch addresses makes the binary code "position independent".

The exceptions are: the RET and RETN instructions that branch to an absolute address that was stored on the stack by an earlier CALL instruction, and the JUMP.pri and CALL.pri instructions that branch to a calculated address.

● Macro instructions

To improve speed and to reduce the memory footprint of the compiled programs, the abstract machine includes several macro instructions. These macro instructions are a sequence of "plain" instructions, in a single opcode. This reduces the memory size, of course, because now a single opcode replaces two or more plain opcodes; it also improves performance, by virtue of reducing the overhead of the P-code decoder inside the abstract machine.

Plain opcodes have zero parameters or one parameter, whereas a macro opcode may have two or more opcodes.

Macro instructions are incompatible with the current JIT run-times —since a JIT run-time removes the instruction decoding overhead, macro instructions have no advantages over plain opcodes when using a JIT run-time. The PAWN compiler can optionally disable the generation of macro instructions.

● Opcode packing

Packed opcodes are instructions where the opcode and its parameter are packed in a single cell. For packed *macro* instructions, only the first parameter is packed in the instruction; the other opcodes are full cells. The concept of packed opcodes is a "space optimization", to reduce the size that a running script takes in memory.

Opcode packing requires token threading, and it is incompatible with the current JIT run-times.

● Native call opcodes

There are two opcodes that are not generated by the PAWN compiler: `SYSREQ.D` and `SYSREQ.ND`. These opcodes are *direct call* variants of `SYSREQ` and `SYSREQ.N` respectively. These opcodes are generated by the abstract machine itself, by run-time patching.

When the script calls a native function, the PAWN compiler generates a `SYSREQ` opcode for the core instruction set and a `SYSREQ.N` opcode when macro instructions are enabled. Both these opcodes cause a jump out of the abstract machine to a routine that handles the dispatching of native functions. You can set up such a routine with `amx_SetCallback`, but there also is a default routine —called `amx_Callback`. The callback/dispatcher function must look up the native function from the parameter of the originating `SYSREQ.*` opcode and then call that native function with the function parameters forwarded. There is a double call in this chain: the `SYSREQ.*` opcode causes a call to the callback* function, which then calls the requested native function.

The `SYSREQ.D` and `SYSREQ.ND` opcodes remove one call, and thereby improve the performance of the native call link. After the callback function has looked up the address of the native function, it *patches* this address right into the code stream of the compiled script, and it changes the `SYSREQ.N` opcode to `SYSREQ.ND` —or `SYSREQ` opcode to `SYSREQ.D` for older systems. The next time this native function is called, there is a new opcode, which calls to the address of the native function directly, bypassing the callback.

This "trick" only works if you use the default callback, or if you implement a similar patching functionality in your custom callback. It also requires that the P-code stream is writeable. If you store the code section of the compiled script in (Flash) ROM, the callback function will be unable to patch the opcodes.

● Cross-platform support

---

\* It is called a *callback* function because it "calls back" into the host application. The host application called the abstract machine and to execute a native function, the flow of control goes back from the abstract machine to the host application.

There is some level of cross-platform support in the abstract machine. Both Big Endian and Little Endian memory addressing schemes are in common use today. Big Endian is the "network byte order", as it is used for various network protocols, notably the Internet protocol suite. The Intel 80x86 and Pentium CPU series use Little Endian addressing.

Multi-byte values are always stored (in the compiled file) in Little Endian. On Big Endian platforms, the abstract machine must convert the instructions from Little Endian to Big Endian.

The abstract machine is optimized for manipulating "cells", 32-bit quantities. Bytes or 16-bit words can only be read or written indirectly, by first generating an address and then use the `LODB.I` or `STRB.I` instructions. The `ALIGN.pri` instruction helps in generating the address.

The abstract machine assumes that when multiple characters are packed in a cell, the first character occupies the highest bits in the cell and the last character is in the lowest bits of the cell. This is how the PAWN language stores packed strings. On a Big Endian computer, the order of the characters is "natural" in the sense that the first character of a pack is at the lowest address and the last character is at the highest address. On a Little Endian computer, the order of the characters is reversed. When accessing the second character of a pack, you should read/write from a lower address then when accessing the first character of the pack.

The PAWN compiler could easily generate the required extra code to adjust the address for each character in the pack. The draw-back would be that a module written for a Big Endian computer would not run on a Little Endian computer and vice versa. So instead, the PAWN compiler generates a special `ALIGN` instruction, whose semantics depend on whether the abstract machine runs on a Big Endian or a Little Endian computer. More specifically, the `ALIGN` instruction does nothing on a Big Endian computer and performs a simple bitwise "exclusive or" operation on a Little Endian computer.

● The "switch" instruction and case table lay-out

The SWITCH instruction compares the value of PRI with the case value in every record in the associated case table and if it finds a match, it jumps to the address in the matching record. The SWITCH opcode has one parameter, which is the address of the case table, relative to CIP (the instruction pointer). At this address, a CASETBL opcode should appear.

Every record in a case table, except the first, contains a case value and a jump address, in that order. The jump address is relative to the address of the record itself. The first record keeps the number of subsequent records in the case table in its first cell and the "none-matched" jump address in its second cell. If none of the case values of the subsequent records matches PRI, the SWITCH instruction jumps to this "none-matched" address. Note again that the first record is excluded in the "number of records" field in the first record.

The records in the case table are sorted on their value. An abstract machine may take advantage of this lay-out to search through the table with a binary search.

# Debugging support

Debugging support comprises two components: the BREAK opcodes and the symbolic information format. The PAWN compiler inserts a BREAK opcode in front of any instruction sequence that starts a statement in the source code. Hence, when a debug hook is set up, the hook function is called before the abstract machine executes the P-code for the statement.

When the debug hook serves as a full (symbolic) debugger, it will then need to browse through the symbolic information for the source files. The compiler attaches the symbolic information to the binary P-code file. The symbolic information consists of a header and several variable sized tables.

| size | 4 bytes | size of the symbolic information chunk |
|---|---|---|
| magic | 2 bytes | signature, must be 0xF1Ef |
| file_version | 1 byte | file format version, currently 8 |
| amx_version | 1 byte | required minimal version of the abstract machine |
| flags | 2 bytes | flags, see below |
| files | 2 bytes | number of entries in the "file table" |
| lines | 2 bytes | number of entries in the "line table" |
| symbols | 2 bytes | number of entries in the "symbol table" |
| tags | 2 bytes | number of entries in the "tag name table" |
| machines | 2 bytes | number of entries in the "machine table" |
| states | 2 bytes | number of entries in the "state table" |

Following the header are, in this order:
⋄ the file table
⋄ the line table
⋄ the symbol table
⋄ the tag name table
⋄ the machine name table
⋄ the state name table

When there are zero entries in any table, the table itself is completely absent from the file. Most tables have variable-length entries, meaning that you have to browse through the symbolic information to locate a specific record.

● The file table

The entries in the file have the following format:

| | | |
|---|---|---|
| `address` | 4-bytes | starting address (relative to COD) |
| `name` | *variable* | zero-terminated string |

The `address` field gives the address in the code segment at which the generated P-code for the file starts. The entire table is sorted on this address field. Given an address relative to the COD pseudo-register, you can look up the file from which the P-code at that address was generated.

P-code from a single file may land on several address ranges in the P-code — through file inclusions, for example. In such case, there are several entries in the file table for the same file.

● The line table

The line table holds line numbers; it is used in combination with the file table:

| | | |
|---|---|---|
| `address` | 4 bytes | starting address (relative to COD) |
| `line` | 4 bytes | line number |

This table maps the addresses in the P-code to line numbers. The file names (relative to which the line numbers are) must be looked up from the file table. The line table is sorted on the `address` field. The `address` is the lowest address at which the generated P-code for the source code line starts. It is common that a BREAK instruction appears on this line.

● The symbol table

Entries for a symbol (variable, function) are more complex:

| | | |
|---|---|---|
| `address` | 4 bytes | address (relative to data or stack) |
| `tag` | 2 bytes | tag for the symbol |
| `codestart` | 4 bytes | starting address (in COD) for scope |
| `codeend` | 4 bytes | ending address (in COD) for scope |
| `ident` | 1 byte | kind of symbol (function/variable) |
| `vclass` | 1 byte | class of symbol (global/local) |
| `dim` | 2 bytes | number of dimensions |
| `name` | *variable* | zero-terminated string |
| *symdim* | *variable* | optional symbol dimension records |

The `address` is relative to either the code segment (COD), the data segment (DAT) or to the frame of the current function —whose address is in the FRM pseudo-register. The `ident` and `vclass` fields indicate to which pseudo-register the address relates, and enables you to locate the value of the symbol.

The possible values for the `ident` field are:

| | |
|---|---|
| 1 | a variable |
| 2 | a "reference", a variable that contains an address to another variable (in other words, a pointer). |
| 3 | an array |
| 4 | a reference to an array (a pointer to an array) |
| 9 | a function |
| 10 | a reference to a function (a pointer to a function) |

The values of the `vclass` field are:

| | |
|---|---|
| 0 | the symbol refers to a global variable (relative to DAT) or to a function (relative to COD) |
| 1 | the symbol refers to a local variable with a stack relative address (relative to FRM; the `address` field may be positive or negative) |
| 2 | the symbol refers to a "static" local variable; the address is not stack relative, but instead relative to DAT |

The `codestart` and `codeend` addresses are relative to the COD pseudo-register. These addresses give the address range in which the symbol is "in scope". For local variables, this is the address range of the compound block in which the variable declaration occurs. Global variables have the ending address set to the last valid P-code address in the file. For functions, the starting and ending address are the address range that the P-code for the function takes. For a function, the fields `address` and `codestart` are equal.

The `tag` field is a numeric tag identifier. You can look up the tag name in the tag table.

When the symbol is an array, or a reference to an array, the `dim` field indicates the number of dimensions, and the number of "symbol dimension" (symdim) records that follow the symbol. When an array has two dimensions, `dim` is 2 and two *symdim* records follow the zero-terminated `name` field. Each symdim record has the format:

| | | |
|---|---|---|
| `tag` | 2 bytes | tag for the array dimension (index) |
| `size` | 4 bytes | size of the dimension (0 if unknown) |

When the "`size`" field of a symdim record is zero, the array size (for that dimension) is indeterminate.

● The `tag` name table

The tag name table enables looking up a tag name from an identifier:

| tag | 2 bytes | tag id |
|---|---|---|
| name | *variable* | zero-terminated string |

A debugger may use the tag name to select an appropriate display format. For example, when a debugger determines that the tag name of a symbol is "`Float:`", it may choose to automatically display the symbol's value as a floating point value, rather than (mis-)interpreting it as an integer.

The "`tags`" table in the "prefix" of the abstract machine (see figure 1 and page 96) also contains a map from tag identifiers to tag names. The table in the prefix area only contains "exported" tags (which may be needed by *any* host program). The table in the debugging information contains *all* tags that are used in the program.

● The automaton table

For scripts that define multiple automaton, the automaton table holds all automaton names. The PAWN compiler also generates a "state variable" per automaton. The address of this variable (relative to the DAT pseudo-register) is also in the automaton table.

| automaton | 2 bytes | automaton id |
|---|---|---|
| address | 4 bytes | address of the "state variable" |
| name | *variable* | zero-terminated string |

● The state table

For each state defined in the script, there is an entry in the state table. Every state relates to an automaton. If no automaton is explicitly defined, the state uses automaton id 0 (zero).

| state | 2 bytes | state id |
|---|---|---|
| automaton | 2 bytes | automaton id |
| name | *variable* | zero-terminated string |

## • Functions

In order to make browsing through debug information easier, the PAWN toolkit comes with support functions that perform the basic tasks of looking up functions, variables and source code line numbers from addresses. These functions are in the files AMXDBG.C and AMXDBG.H. These functions are provided as "example implementations" —you may want (or need) to adapt them to your environment.

# Code generation notes

The code generation of the PAWN compiler is fairly straightforward (also due to the simplicity of the abstract machine). A few points are worth mentioning:

◇ For a native function, the PAWN compiler generates a SYSREQ instruction instead of the normal function call. The parameter of the SYSREQ instruction is an index in the native function table. A function in PAWN cleans up its arguments that were pushed on the stack, because it returns with the RETN instruction. The SYSREQ instruction does not remove items from the stack, so the PAWN compiler does this explicitly with a STACK instruction behind the SYSREQ instruction.

The arguments of a native function are pushed on the stack in the same manner as for a normal function.

In the "PAWN" implementation of the abstract machine (see ), the "system request" instructions are linked to the user-installed callback function. Thus, a native function in a PAWN program issues a call to a user-defined callback function in the abstract machine.

◇ At a function call, a PAWN program pushes the function arguments onto the stack in reverse order (that is, from right to left). It ends the list of function arguments on the stack by pushing the number of bytes that it pushed to the stack. Since the PAWN compiler only passes cell-sized function arguments to a function, the number of bytes is the number of arguments multiplied by the size of a cell.

A function in PAWN ends with a RETN instruction. This instruction removes the function arguments from the stack.

◇ When a function has a "reference" argument with a default value, the compiler allocates space for that default value on the heap.

For a function that has an array argument with a default value, the compiler allocates space for the default array value on the heap. However, if the array argument (with a default value) is also const, the PAWN compiler passes the default array directly (there is no need to make a copy on the heap here, as the function will not attempt to change the array argument and, thereby, overwrite the default value).

◇ The arguments of a function that has "variable arguments" (denoted with the `...` operator, see the PAWN booklet "The Language") are always passed by reference. For constants and expressions that are not *lvalues*, the compiler copies the values to a cell that is allocated from the heap, and it passes the address of the cell to the function.

◇ For the "`switch`" instruction, the PAWN compiler generates a SWITCH opcode and a case table with the CASETBL opcode. The case table is generated in the COD segment; it is considered "read-only" data. The "none-matched" address in the case table jumps to the instruction of the `default` case, if any.

Case blocks in PAWN are not drop through. At the end of every instruction in a `case` list, the PAWN compiler generates a jump to an "exit" label just after the `switch` instruction. The PAWN compiler generates the case table between the code for the last `case` and the exit label. By doing this, every case, including the `default` case, jumps around the case table.

◇

Multi-dimensional arrays are implemented as vectors that hold the offsets to the sub-arrays. For example, a two-dimensional array with four "rows" and three "columns" consists of a single-dimensional array with four elements, where each element is the offset to a three-element single-dimensional array. The total memory footprint of array is $4 + 4 \times 3$ cells. Multi-dimensional arrays in PAWN are similar to pointer arrays in $C/C^{++}$.

As stated above, the "major dimension" of multi-dimensional arrays holds the offsets to the sub-arrays. This offset is in bytes (not in cells) and it is relative to the address of the cell from which the offset was read. Returning to the example of a two-dimensional array with four rows and three columns (and assuming a cell size of four bytes), the memory block that is allocated for the array starts with the four-cell array for the "rows", followed by four arrays with each three elements. The first "column" array starts at four cells behind the "rows" array and, therefore, the first element of the "rows" array holds the value $4 \times cellsize$ (16 for a 32-bit cell). The second column array starts at three cells behind the first column array, which is seven cells behind start of the rows array. The offset to the second column array is stored in the it second element of the rows array, and the offset of the second column relative to the second cell of the rows array is *six* cells. The second value in the rows array is therefore $6 \times cellsize$.

For a specific example, assume an array that is declared as:

```
new values[4][3] = [ [ 1, 1, 1 ],
                     [ 2, 2, 2 ],
                     [ 3, 3, 3 ],
                     [ 4, 4, 4 ] ]
```

The sequence of values in memory for this array, where a "$c$" suffix on a number means that the value should be scaled for the size of a `cell` in bytes, is:

$4_c$, $6_c$, $8_c$, $10_c$, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4

For a three-dimensional array, the entries in the vector for the major dimension hold the offsets to vectors for each minor dimension. The vector tables for all dimensions come in front of the contents of the array.

◇ The destructor operator takes an array with a single dimension on input, and this array holds all elements of a variable that must be destructed:
  • For simple variables, the variable is passed by reference, which makes it appear as an array with one element.
  • For arrays with one dimension, the array is passed without modification
  • For arrays with two or more dimensions, the destructor operator receives the address *behind* the "indirection tables" for the major dimensions. As documented above, a multi-dimensional array starts with vectors for the major dimensions that each hold the offsets to the dimension below itself. The data for the array itself is packed behind these offset arrays. By passing the address where the array data starts, the destructor operator can access the array elements as if it were an array with a single dimension.

◇ As of version 2.0, the PAWN compiler puts a `HALT` opcode at the start of the code (so at code address 0). Before jumping to the entry point (a function), the abstract machine pushes a zero return address onto the stack. When the entry point returns, it returns to the zero address and sees the `HALT` instruction.

◇ The `sleep` instruction generates a `HALT` opcode with the error code 12 ("sleep"). When the abstract machine sees this special error code, it saves the state of the stack/heap (rather than resetting it), in order to be able to restart the abstract machine.

◇ The PAWN compiler adds special comments to the assembler file (with the forms ";$exp" and ";$par") to aid the peephole optimizer to make the correct decisions. These comments mark the end of an "expression statement" or the end of a function parameter. The code generated by the compiler does not carry the value of a register from one statement/expression to another, and the peephole optimizer uses this information to avoid saving registers whose values will not be used again anyway.

◇ For functions that have states, the compiler creates a jump table at the function address; each entry jumps to a specific implementation. Technically, the table is a "case table" similar to the one used for a "`switch`" statement and it indeed uses the SWITCH opcode to jump to the correct implementation. The "default" case points to the fall-back function, or to an error exit point if a fall-back function was absent.

As an aside, this schema is more efficient than a hand-coded `switch` on a state variable, because a hand-coded `switch` would need to reside inside a function of its own, using an extra function frame and using extra stack to store another return address and to forward and parameters.

# Adding a garbage collector

PAWN uses only static allocation for all of its objects. The advantage of static allocation is that the memory requirements of a PAWN script are easy to determine (the PAWN compiler does this with the `-d2` option), and that the memory footprint and run-time performance become fully deterministic.

That non-withstanding, for dealing with dynamically sized objects in PAWN, a garbage collector is very convenient. This appendix describes how a garbage collector can be added to a host application that uses the PAWN toolkit. It is implemented as a separate library.

## • How to use

The purpose of the garbage collector is to notify your program of objects that are no longer in use and that can, hence, be released. To this end, the garbage collector needs a data structure to register objects that are *in use* and it needs a way to notify the host application (your program) of redundant objects. These two elements must be initialized before you start using the garbage collector.

The data structure that records objects that are "in-use" is a hash table. Its size must be a power of two —in fact, the parameter that you pass to function `gc_settable` is the "power". That is, passing 10 as the argument to `gc_settable` creates a hash table that holds $2^{10}$, or 1024, items. There is a low bound on the size of 128 elements, meaning that the exponent parameter must be at least 7. The maximum size of the hash table is the maximum value of a signed integer: 32,767 for 16-bit platforms and 2,147,483,648 for 32-bit platforms (the maximum exponent is 15 or 31 for 16-bit and 32-bit platforms respectively). The second parameter to `gc_settable` is a collection of flags. The only flag defined at this writing is `GC_AUTOGROW`, which tells the garbage collector that it may automatically increase the size of the hash table when it becomes full.

For every object that is no longer referred to in any abstract machine that was scanned, the garbage collector calls a callback function to release it. But first, you have to register this callback function, of course. This, you do with function `gc_setcallback`.

By intent, the signature for the callback function has been made compatible with the standard C function `free`. If your host program allocates its objects with `malloc`, then you may be able to set the standard `free` function as the garbage

collector callback. If you need additional clean-up code, or if you do not allocate the objects with `malloc`, you have to write an appropriate callback.

Once the hash table and the callback are set, your host program (or your native function library) can allocate objects and mark them as "used" with the function `gc_mark`. The value that you pass in must a *non-zero* value that uniquely identifies the object, and it must be a "`cell`" data type —the data type of the PAWN language. If the size of a pointer is the same as that of a `cell` (or smaller), you can mark a pointer to an object (by simply casting it as a `cell`). Other mechanisms are that you allocate the object from a list that you maintain internally, and "mark" the index to the object in this list. It is important that you mark exactly *the same* value as what the native function returns to the PAWN script.

Once every while, on a timer or at any other moment that is convenient, the host program should call `gc_scan` once or multiple times, followed by a single call to `gc_clean`. Before `gc_clean` finishes, it invokes the callback function for every object that is no longer referenced. The parameter to the callback function is the same value that you have passed to `gc_mark` for the function. Function `gc_scan` detects "live" objects, function `gc_clean` cleans up any object that is not alive.

A host application may run multiple scripts concurrently, and it may therefore have multiple abstract machines in existence at any time. The garbage collector collects the object references for all objects that were allocated for all abstract machines. When performing a garbage collection run, the program should scan all abstract machines (using `gc_scan`) and finish with a single call to `gc_clean`. When an abstract machine disappears, all objects allocated to that abstract machine (that are not referred to by other abstract machines) are cleaned up in the subsequent garbage collection run —simply because `gc_scan` is not called on the abstract machine that is gone.

At the end of the program, call `gc_settable` with size zero. Earlier I wrote that there is a lower bound on the input value to `gc_settable` of 7, but the value zero is a special case. As an aside, `gc_settable` calls `gc_clean` internally when the table exponent is zero, to dispose any remaining object in the table.

## • Rescaling the garbage collector

The garbage collector is built on a hash table, which is allocated dynamically. A hash table is a data structure that allows quick look-up. It does this by calculating an index value from some arbitrary property of the object and it stores a reference to the object at that calculated index in the table. For the garbage collector, the

index is calculated from the "`value`" parameter that you pass into the function `gc_mark`.

A hash table should not be too small —because it can store no more objects than fit in the table, and it should not be too large, as that would waste memory and decrease performance. The garbage collector makes the table size adjustable: you can start running with a small table and grow it on an "as needed" basis. If desired, you may also shrink the hash table. Growing or shrinking the hash table preserves the objects currently in the table.

A problem with hash tables in general is that of "collisions": two different objects may get the same index in the hash table. There are various strategies of coping with this situation; the garbage collector uses the simplest one: "probing". If a collision occurs, the new object is not stored at its calculated index, but at a fixed offset from the calculated index. To avoid clusters in the table, the offset decreases from roughly a quarter of the table size (except for tables exceeding 64 kiB) down to 1; to avoid "blind spots" in the table, the probing offset is always a prime number.

When the hash table is full, `gc_mark` may first attempt to grow the table (depending on whether the `GC_AUTOGROW` was set in the call to `gc_settable`). It returns with an error code if growing the table fails or if it is not permitted. The host program can then do a garbage collection run, in the hope that this frees up some slots in the hash table; the host program may also attempt to grow the hash table itself. As the hash table is allocated dynamically, the attempt to resize it may also fail. The end result is that `gc_mark` may fail and your host program has no way to recover from it.

Unrecoverable failure of `gc_mark` can be avoided, though: instead of waiting for a full table to happen, a host program can decide to grow the table well before it becomes full. If that fails, `gc_mark` still succeeds and the next few calls to `gc_mark` will also succeed. Hence, the host application has the opportunity to free up memory or inform the user of "low memory" —a message that is friendlier than one like "out of memory, cannot continue".

There is another reason why early growing of the hash table is a good strategy: performance. Linear probing is a simple method for coping with collisions, but it also leads to heavily degraded performance once the hash table fills up. It is probably best when the hash table usage does not exceed 50%. The function `gc_tablestat` returns the current "load" of the hash table, in a percentage of its size.

## • An example implementation

To use the garbage collector in an example, we must first have a native function library that creates garbage. For this example, I have chosen the "Better String library" by Paul Hsieh, a library that enables working with dynamically allocated variable length strings in C/C++.

The first step is to create wrapper functions for a subset of the library. For the purpose of demonstrating the garbage collector, I have chosen a *minimal* subset, just enough to run the example program below —in real applications you would add significantly more functions:

```
#include <bstring>

main()
    {
    new String: first = bstring("Hello")
    new String: second = bstring("World")

    new String: greeting = first + bstring(" ") + second

    new buffer[30]
    bstrtoarray .target = buffer, .source = greeting
    printf buffer
    }
```

Two primary native functions implemented below perform a conversion to or from PAWN arrays: `n_bstring` and `n_bstrtoarray`. Conversion from an array to the "`bstring`" type (of the Better String library) is needed to handle literal strings; the conversion back to a PAWN array is needed because the native functions in the "console I/O" extension module do not support the `bstring` type. Again, in practice you would probably modify the `printf` and other native functions to work with `bstring`, so that converting back to PAWN arrays is never necessary.

```
#define VERIFY(exp)     do if (!(exp)) abort(); while(0)

static cell AMX_NATIVE_CALL n_bstring(AMX *amx,const cell *params)
        /* native String: bstring(const source[] = ""); */
{
  cell hstr = 0;
  char *cstr;

  amx_StrParam(amx, params[1], cstr);
  if (cstr != NULL) {
    hstr = (cell)cstr2bstr(cstr);
    VERIFY( gc_mark(hstr) );
  } /* if */
  return hstr;
}
```

```
static cell AMX_NATIVE_CALL n_bstrtoarray(AMX *amx,const cell *params)
        /* native bstrtoarray(target[], size = sizeof target,
         *                     String: source, bool: packed = false);
         */
{
  char *cstr = bstr2cstr((const bstring)params[3], '#');
  int length = strlen(cstr) + 1;
  cell *cellptr;

  if (params[4])
    length *= sizeof(cell);
  if (params[2] >= length) {
    cellptr = amx_Address(amx, params[1]);
    amx_SetString(cellptr, cstr, params[4], 0);
  } /* if */
  free(cstr);
  return 0;
}

static cell AMX_NATIVE_CALL n_bstrdup(AMX *amx,const cell *params)
        /* native String: bstrdup(String: source); */
{
  cell hstr = (cell)bstrcpy((const bstring)params[1]);
  VERIFY( gc_mark(hstr) );
  return hstr;
}

static cell AMX_NATIVE_CALL n_bstrcat(AMX *amx,const cell *params)
        /* native String: bstrcat(String: target, String: source); */
{
  cell hstr = params[1];
  bconcat((bstring)hstr, (const bstring)params[2]);
  return hstr;
}
```

The wrapper functions that allocate new `bstring` instances are different from common wrapper functions in that they call `gc_mark`. Note that the wrapper functions that do not create *new* `bstring` instances do not need to mark an object to the garbage collector.

Error checking is primitive in this example. When the garbage collector's hash table is full and it cannot grow, the program simply aborts. As discussed in a preceding section, it is advised to grow the table well before it would become full.

Now we must modify the host application to set up the garbage collector. In my case, this is an adapta

Initializing the garbage collector is an easy step, because the memory de-allocator for the "Better String library" is compatible with the callback function of the

garbage collector. All one has to do is to insert the following lines somewhere before the call to `amx_Exec`:

```
gc_setcallback((GC_FREE)bdestroy);
gc_settable(7, GC_AUTOGROW);   /* start with a small table */
```

Cleaning up the garbage collector before exiting is easy too:

```
gc_settable(0, 0);        /* delete all objects and the hash table */
```

The harder part is running the garbage collector at *appropriate* times. On one hand, you will want to call the garbage collector regularly, so that the table does not contain too much "garbage"; on the other hand, calling the garbage collector too often decreases the overall performance. Actually, it would be best if the collector ran at times that CPU usage is low.

Even if we just wish to call the garbage collector on a regular interval, a minor problem is that there is no portable way of doing so. In Linux and UNIX, you may use the `signal` and `alarm` functions and in Microsoft Windows the `SetTimer` function may be of use. Multi-threading is another option, but be aware that you have to implement "mutual exclusion" access yourself (e.g. with semaphores, or a critical section).

The function that performs a garbage collection run may be like the one below. The function expects the abstract machines to scan in an array. It grows the hash table when its usage exceeds 50%.

```
void garbagecollect(AMX amx[], int number)
{
  int exp, usage, n;

  /* see whether it may be a good time to increase the table size */
  gc_tablestat(&exp, &usage);
  if (usage > 50) {
    if (gc_settable(exp+1, GC_AUTOGROW) != GC_ERR_NONE)
      fprintf(stderr, "Warning, memory low\n");
  } /* if */

  /* scan all abstract machines */
  for (n = 0; n < number; n++)
    gc_scan(&amx[n]);

  /* clean up unused references */
  gc_clean();
}
```

With the goal of providing a complete example that compiles and runs on all platforms* that the PAWN toolkit currently supports, I have "hooked" function `garbagecollect` (implemented above) onto the debug hook. That is, the host application sets up a debug hook and the debug hook function calls `garbagecollect` on various events. Doing this in anything other than a demo program is *not* advised, for several reasons:

◇ The debug hook can only monitor a single abstract machine, whereas you are likely to have multiple concurrent abstract machines in real projects.

◇ The debug hook will never called when the script was compiled without debug information.

That behind us, below is a debug hook that calls the garbage collector, after executing every 100 lines.

```
int AMXAPI aux_Monitor(AMX *amx)
{
  static int linecount;

  if (--linecount > 0)
    return AMX_ERR_NONE;
  linecount = 100;
  garbagecollect(amx, 1);

  return AMX_ERR_NONE;
}
```

## • Other notes

As discussed earlier, the `gc_clean` function invokes the callback function to free any object that is no longer in use in any abstract machine that was scanned. The function assumes that the callback indeed frees the object: it will not report it again.

Each object should only be in the hash table once. If you call `gc_mark` with a value that is already in the hash table, the function returns an error. It is a non-fatal error, but nevertheless it is better to avoid adding the same pointer/object twice to the garbage collection table.

The probing algorithm used by the garbage collector differs from both the well known linear and quadratic probing algorithms, but its properties (related to clustering or "clumping") are similar to those of quadratic probing.

---

* The standard distribution comes with the source code for a minimal host application, in the subdirectory "amx/pawnrun/" of where the toolkit was installed.

The design of a good hash function/equation is another recurring theme in research. As the garbage collector discussed here is general purpose, nothing about the input key (the parameter to `gc_mark`) may be assumed. The hash generation algorithm used in the garbage collector "folds" the key value a few times, depending on the size of the "cell" and the size of the hash table. Folding means that the key value is split in half and the two halves are combined with an "exclusive or" operation. Concretely, if the hash table exponent (the first parameter to `gc_settable`) is less than, or equal to 16, a 32-bit key value is first split into two 16-bit values and then the upper half is "exclusive or'ed" to the first half, resulting in a single 16-bit value —the new key. When the table exponent is less than, or equal to 8, the folding occurs twice.

Frequently, the origin of the key value is a pointer. In typical memory managers, the lowest bits are fixed. For example, it is typical that memory is allocated at an address that is a multiple of 8 bytes, to ensure optimal alignment of data. The hash table function attempts to copy with this particular aspect by swapping all bits of the least-significant byte.

# Encrypted scripts

In some circumstances, you may wish to protect the compiled scripts that you create —that is: to make disassembly of the P-code file hard to do, *very* hard.

The PAWN compiler optionally creates encrypted P-code files using the "KeeLoq" algorithm and a 64-bit key that you specify on the command line. The cipher works for 32-bit and 64-bit cells. The source code of the KeeLoq encryption and decryption routines are provided with the PAWN source code.

What remains to be done, by you, is to add the KeeLoq decryption to your implementation of the abstract machine. The simplest case is when the scripts do *not* use overlays, in which case you can decrypt the entire script before calling `amx_Init`. The compiler sets the flag `AMX_FLAG_CRYPT` when it has generated an encrypted P-code file. The file header is not encrypted, and therefore, this part must skipped in decrypting. Referring to function `aux_LoadProgram` on , between steps 3 and 4, you should add:

```
if ((hdr.flags & AMX_FLAG_CRYPT) != 0)
   decrypt(memblock + hdr.cod, hdr.size - hdr.cod, SecretKey);
```

The variable `SecretKey` is a 64-bit key, which must of course be the same as the key that was passed to the PAWN compiler. The decryption routine loops over the memory block and calls the KeeLoq cipher for each 32-bit word.

```
#include "keeloq.h"

void decrypt(void *buffer, int size, uint64_t key)
{
  uint32_t *ptr = (uint32_t*)buffer;
  while (size > 0) {
    *ptr = KeeLoq_Decrypt(*ptr, key);
    ptr++;
    size -= sizeof(uint32_t);
  } /* while */
}
```

For the function `KeeLoq_Decrypt`, please see the source code provided with the PAWN abstract machine.

For scripts that you overlays, only the data section should be decrypted before calling `amx_Init`. The overlay blocks with code must be decrypted be the overlay callback. See appendix J on for details on overlays.

# Running scripts from ROM

The default configuration of the PAWN Abstract Machine assumes that the P-code runs from RAM. Various functions modify (or *patch*) the P-code after it is loaded. To run a compiled script directly from ROM, you need to build the Abstract Machine with a few specific options and you also need to compile the PAWN source code in a particular way.

The run-time patching of native function calls is incompatible with ROM execution, and it must be disabled. See the discussion at for the patching that occurs with native function calls. The compile-time flag to disable run-time patching is `AMX_DONT_RELOCATE`.

When compiling with GNU GCC or when using an assembler kernel for the abstract machine, it is advisable to force "token threading" by defining the macro `AMX_TOKENTHREADING`. Token threading is compatible with ROM execution, unlike the alternative "direct threading" threading implementation that PAWNcan use. If support for packed opcodes is left enabled, token threading is also the default of the abstract machine. When you disable the support for packed opcodes (option `AMX_NO_PACKED_OPC`), the abstract machine will use "direct threading" (which is incompatible with ROM execution) unless token threading is explicitly set.

In addition, the indirection table that matches native functions to a script cannot be altered (it is in ROM too). As explained on , the solution is to use a native table with hardcoded unique (negative) numbers for the native functions. Since the numbers are hardcoded, function `amx_Register` is redundant.

In the callback function (for native functions), you will then need to map the unique negative numbers to function pointers. A simple way to do this is to use an array, where the unique number serves as the index —after taking its absolute value. This table-based lookup avoids any name comparison or (binary) table search. To use table-based lookup, the macro `AMX_NATIVETABLE` must be set to the name of a global variable that holds the function addresses. For details, see

and further.

Function `amx_Init`, the first function to call on an abstract machine, receives a pointer to the start of the header and the P-code. Unless told otherwise, `amx_Init` considers that the data and stack areas follow the P-code immediately. In the case of running from ROM, the P-code and the combined data/stack sections must reside in different blocks: the data/stack section must be in RAM and the

P-code is in ROM. To configure the abstract machine as such, store the address of a RAM block of suitable size in the `data` field of the `AMX` structure prior to calling `amx_Init`, and pass a pointer to the P-code in ROM as the second parameter to `amx_Init`.

```
unsigned char amx_dataseg[DATASIZE];    /* data + stack */
extern const unsigned char amx_pcode[]; /* P-code in ROM */

AMX amx;
memset(&amx, 0, sizeof amx);
amx.data = (unsigned char*)amx_dataseg;
error = amx_Init(&amx, amx_pcode);
```

Function `amx_Init` will copy the contents of the data section of the script in ROM to the pointer in the `data` field. This is necessary because the data section of the script contains string literals and other initialized static and global data. Optionally, you can also skip this step: by adding the constant `AMX_FLAG_DSEG_INIT` to the `flags` field of the AMX structure prior to calling `amx_Init`, the data section will not be initialized —leaving that for you to do explicitly. For example:

```
memset(&amx, 0, sizeof amx);
amx.data = (unsigned char*)amx_dataseg;
amx.flags = AMX_FLAG_DSEG_INIT;
error = amx_Init(&amx, amx_pcode);

/* add code to initialize amx_dataseg */
```

The PAWN compiler must also be configured to generate P-code that is suitable for execution from ROM. Since the Abstract Machine is compiled with hardcoded indices for native functions, the include file(s) for the PAWN compiler must declare all native functions with hardcoded indices.

Other configurations for the compiler are best set in an include file. The PAWN compiler always tries to include the file `default.inc` implicitly and before any other include file. It has become common practice to let `default.inc` include any other "standard" include files and to add configuration options.

In order to get small compiled "smallcapsamx" files, we strip off as many unneeded tables as we can. Part of these are the names of the native functions; this was already explained above, in the discussion of the native function table implementation in the abstract machine. The table with library names can be removed completely. The library table is used for dynamically loading extension modules, on operating systems that support this (Microsoft Windows, Linux), but it is useless on embedded operating systems. To turn the library table generation off, put the following pragma in an include file (e.g. `default.inc`):

```
#pragma library -
```

It is a good idea to also tell the PAWN compiler how much memory a compiled script may take at run time. This is controlled by the pragmas `amxlimit` and `amxram`. For example:

```
#pragma amxlimit 8192        /* 8 kiB available for code */
#pragma amxram   4096        /* 4 kiB available for data + stack */
```

# Running scripts with overlays

If your system has a (read-only) backing store of sufficient size, overlays enable scripts with a large code footprint to run in little RAM memory. An overlay system is akin to virtual memory, but for code only (no data) and with compiler support. Advantages of overlays are that they require only read access to a mass storage medium (so this can be ROM) and that it does not impact run-time performance as much as true virtual memory systems. A disadvantage of overlays is that there is still a limit on the data and stack portions of the script.

To use overlays, the following steps should be taken:

◇ The script must be compiled with overlay support. See below and see the "Language Guide" for details —briefly: you have to use the option "-V" on the compiler.

◇ In the call to `amx_Init`, you have to initialize a few fields in the AMX structure. Unless told otherwise, `amx_Init` assumes that the AMX header, the code section and the data section are consecutively in a single memory block. The reason for using overlays, though, is that you cannot provide a single memory block with the required size.

◇ You have to implement a helper function that loads overlays from the backing store into memory. That function must also adjust two fields in the AMX structure.

● Configuring the compiler

The `-V` option allows you to set the maximum allowable size of an overlay. As this value depends on the implementation, you may want to "fix" that value. One option is to add a conditional definition in the standard include file, as in:

```
#if overlaysize > 0
    #pragma overlaysize 2040    // set the maximum overlay size of 2040
#endif
```

The trick used here is that if overlays are enabled, the predefined constant `overlaysize` is always greater than zero. When the user adds the option "-V" to the command line (without setting a size), `overlaysize` is set to 1. The pragma overrules the standard value (i.e. "1") or any value that the user specified with the implementation-defined value.

Note that the value set for the pragma "`overlaysize`" must be the largest overlay that will fit in the available memory of the overlay manager. If the overlay manager has overhead for an allocation, this "overhead" value must subtracted from the `overlaysize`. Notably, the overlay manager included in the PAWN distribution has an overhead of 8 bytes for a typical 32-bit system. If you set aside a memory block of 2048 bytes for this overlay manager, the maximum value of the pragma `overlaysize` is 2040.

An alternative configuration is that the implementation has a maximum size for the total of the overlay pool, the data and the heap plus stack. In this case, set the pragma `amxlimit` to that total size and set `overlaysize` to 1 (if overlays are to be used). The maximum size for an overlay then depends on the size of the data and of the heap and stack.

## • Initializing the abstract machine

The code snippet below is typical for initializing an abstract machine with overlays. Instead of reading the entire file in a single memory block, the snippets prepares three areas: one for the header, one for the combined data/heap/stack and a swap region for overlays. This may technically reside in a single memory block, but that is not required.

```
/* assume the file to a ".amx" file is already opened and that its
 * handle is in fpAmx. Also assume that "header", "datablock" and
 * "overlaypool" are pointers to memory blocks of sufficient size.
 */

/* read the AMX header */
rewind(fpAmx);
fread(header, 1, hdr.cod, fpAmx);

/* read the data section (the memory block should have sufficient
 * size to also hold the heap and stack)
 */
fseek(fpAmx, hdr.dat, SEEK_SET);
fread(datablock, 1, hdr.hea - hdr.dat, fpAmx);

/* initialize the overlay pool (size 2048 bytes) */
amx_poolinit(overlaypool, 2048);

/* initialize the abstract machine */
memset(amx, 0, sizeof *amx);
amx->data = datablock;
amx->overlay = prun_Overlay;
result = amx_Init(amx, header);
```

In the above snippet, the `data` field of the `AMX` structure is set to a specific value, and it was then read from file. The complete AMX header of the compiled script (including the function, name and overlay tables) is also read, in a separate memory block.

The PAWN toolkit comes with an implementation of a "memory pool" with a "least revently used" queue. The function `amx_poolinit` comes from this module. You may choose to use (or implement) a different overlay manager. The basic requirement is that it should never fail, provided that the total memory size is sufficiently big for the largest overlay. Memory fragmentation should never be a reason for the overlay manager to fail.

• The overlay callback function

The previous code snippet sets a callback function for loading and managing overlays, here called `prun_Overlay` (you may of course choose a different name). The callback controls the loading and the memory allocation. In a simple case, the function could look like the one below:

```
int AMXAPI prun_Overlay(AMX *amx, int index)
{
  AMX_HEADER *hdr;
  AMX_OVERLAYINFO *tbl;

  hdr = (AMX_HEADER*)amx->base;
  tbl = (AMX_OVERLAYINFO*)(amx->base + hdr->overlays) + index;
  amx->codesize = tbl->size;
  amx->code = amx_poolfind(index);
  if (amx->code == NULL) {
    amx->code = amx_poolalloc(tbl->size, index);
    fseek(fpAmx, (int)hdr->cod + tbl->offset, SEEK_SET);
    fread(amx->code, 1, tbl->size, fpAmx);
  } /* if */
  return AMX_ERR_NONE;
}
```

The function assumes that the file to the compiled script is (still) open and that its handle is in a global variable called "`fpAmx`". The function first checks whether the index for the overlay is valid. Then it checks whether the overlay is already in memory. The overlay manager can keep multiple overlays in memory concurrently; if an overlay is already in memory, there is no need to reload it. If the overlay is not in memory, the function allocates a block of a suitable size and loads the relevant portion of the compiled script into that block.

The overlay callback must also adjust two fields in the AMX structure: `code` and `codesize`. The field `code` must point to the start of the overlay. Since each

overlay is a separate (and complete) function, the `code` field points to the start of the function after loading the overlay into memory. The `codesize` field is the size, in bytes, of the overlay. The abstract machine uses this field for detecting stack overruns and other troubles. Note that the overlay callback must adjust these fields even if the overlay function is already in memory.

To better present the essence of the code, no error checking is present in the function presented above. On a well configured system, overlay loading should never fail. However, to guard against a script that was compiled with an incorrect value for the maximum overlay size, you may want to add a check whether `amx_poolalloc` returns `NULL`. For instance, replace the call to `amx_poolalloc` with:

Alternatively, an implementation can browse through all overlays to check their size before calling amx_Init

```
if ((amx->code = amx_poolalloc(tbl->size, index)) == NULL)
    return AMX_ERR_OVERLAY;   /* failure allocating memory for the overlay */
```

Function `amx_Init` calls the overlay callback (called "`prun_Overlay`" in this appendix) for every overlay in the compiled script. Any overlay that is too big for the pool is thus detected before `amx_Init` returns.

### • Loading scripts with and without overlays

In your system, you may want to give the end user the option of whether to run their scripts with overlays or without overlays. Scripts with overlays *always* run slower than scripts without overlays, because of the overhead in the overlay loading. So you could give the script writers the choice between performance and memory.

When loading the compiled script, you must then verify whether the script uses overlays, by checking the `AMX_FLAG_OVERLAY` bit in the `flags` field in the formatted part of the AMX header. If this bit is set, proceed with loading/initializing the script as described in this appendix; if cleared, use the procedure on .

# License

The **software toolkit** "PAWN" (the compiler, the abstract machine and the support routines) is copyright © 1997–2012 by ITB CompuPhase, and distributed under the "Apache License" version 2.0, which is reproduced below, plus an exception clause regarding static linking. See the file NOTICES for contributions and their respective licenses.

**EXCEPTION TO THE APACHE 2.0 LICENSE**

As a special exception to the Apache License 2.0 (and referring to the definitions in Section 1 of this license), you may link, statically or dynamically, the "Work" to other modules to produce an executable file containing portions of the "Work", and distribute that executable file in "Object" form under the terms of your choice, without any of the additional requirements listed in Section 4 of the Apache License 2.0. This exception applies only to redistributions in "Object" form (not "Source" form) and only if no modifications have been made to the "Work".

**TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION**

1. **Definitions**

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally

submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. **Grant of Copyright License.** Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. **Grant of Patent License.** Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. **Redistribution.** You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   a. You must give any other recipients of the Work or Derivative Works a copy of this License; and

   b. You must cause any modified files to carry prominent notices stating that You changed the files; and

   c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications,

or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. **Submission of Contributions.** Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. **Trademarks.** This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. **Disclaimer of Warranty.** Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. **Limitation of Liability.** In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. **Accepting Warranty or Additional Liability.** While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

❖

# Index

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