the Small booklet

Implementor's Guide

February 2005

ITB CompuPhase

"Java" is a trademark of Sun Microsystems, Inc.
"Microsoft" and "Microsoft Windows" are registered trademarks of Microsoft Corporation.
"Linux" is a registered trademark of Linus Torvalds.
"CompuPhase" is a registered trademark of ITB CompuPhase.

Copyright © 1997–2005, ITB CompuPhase; Eerste Industriestraat 19–21, 1401VL Bussum, The Netherlands (Pays Bas); telephone: (+31)-(0)35 6939 261
e-mail: info@compuphase.com, WWW: http://www.compuphase.com

The information in this manual and the associated software are provided "as is". There are no guarantees, explicit or implied, that the software and the manual are accurate.

Requests for corrections and additions to the manual and the software can be directed to ITB CompuPhase at the above address.

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

Introduction

"SMALL" 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 subtitle "The Language". This "Implementor's Guide" discusses how to embed the SMALL scripting language in a host application.

The SMALL 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"). SMALL 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 SMALL in host applications that are not written in C or C++, I suggest that you use the AMX DLLs under Microsoft Windows.

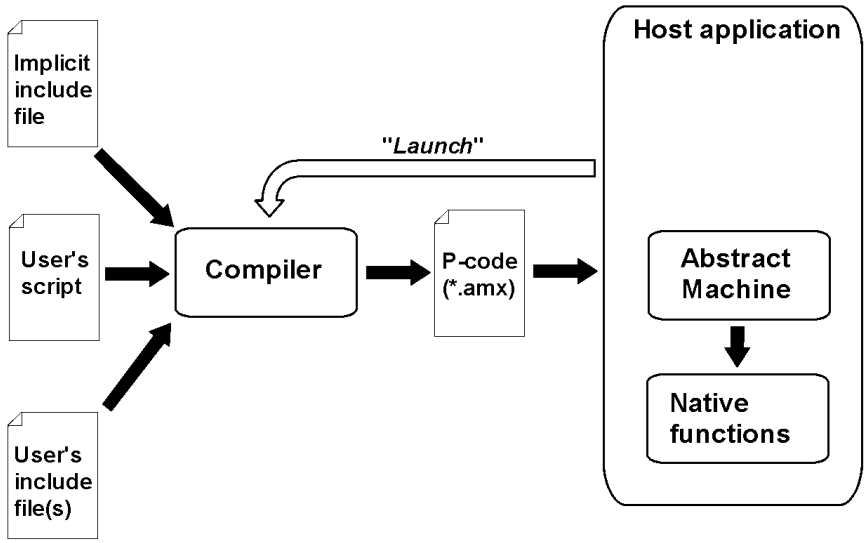
There is a short chapter on the compiler. Most applications execute the compiler as a standalone utility with the appropriate options. Even when you link the compiler into the host program, its API is still based on options as if they were specified on the command line.

The abstract machine is a function libary. 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.

Gluing Small to your product

The SMALL 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 SMALL toolset must be *integrated* with the product.



The two main parts of the SMALL 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 SMALL 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 SMALL 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 SMALL 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 operloaded 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 SMALL 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 SMALL 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 SMALL compiler and the abstract machine respectively. The most common setup is the one where the compiler runs as a separate process that is spawned from the host application.

The compiler

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

Deployment / installation

In most operating systems, the compiler consists of a single 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.

Concretely, to set up the SMALL compiler on a system:

- ◇ copy the program file for the compiler (typically "`sc`" for Linux/Unix and "`sc.exe`" for Microsoft Windows) in a directory of your choice;
- ◇ optionally copy or create a configuration file, called "`sc.cfg`", in the same directory;
- ◇ add a subdirectory called "`include`" below the directory in which the compiler and `sc.cfg` reside, and copy the include files into that directory —especially add the "DEFAULT.INC" prefix file into that directory, if applicable. For details on the prefix file, look up the compiler command line option `-p` in the SMALL booklet "The Language".

Configuration file: 4

Prefix file: 2

Configuration file

On platforms that support it (currently Microsoft DOS, Microsoft Windows and Linux), the compiler reads the options in a "configuration file" on startup. The configuration file must have the name "`sc.cfg`" and it must reside in the same directory as the compiler executable program.

In a sense, the configuration file is an implicit response file (see the SMALL 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 SMALL 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).

Run-time errors: 60

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.

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, 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, 0);
  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 SMALL 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 SMALL 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 SMALL program may also abort the SMALL 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 9 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 SMALL 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;

3 reads in the complete file;
4 sets up an optional hook for a debugger or run-time "monitor" function
5 cleans up resources that it allocated in case an error occurs.

```
int aux_LoadProgram(AMX *amx, char *filename, void *memblock,
                    int AMXAPI (*amx_Debug)(AMX*))
{
  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);
  amx_SetDebugHook(amx, amx_Debug);     /* set up the debug hook */
  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: SMALL 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`. If a debugger hook must be set up for the abstract machine, the relevant statement appears between `memset` and `amx_Init`.

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 instructions. 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 `PrintUsage` 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 line %ld\n",
         errorcode, aux_StrError(errorcode),
         (amx != NULL) ? amx->curline : 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 SMALL programs. A drawback, however, is that the SMALL program runs uncontrolled once it is launched with `amx_Exec`. If the SMALL program enters an infinit 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 SMALL 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 abstract machine first checks queries the debug hook whether it want to receive further events. The debug hook *must* return an acknowledging value on this initial call.

To install a debug hook, call `amx_SetDebugHook` before calling `amx_Init`. For an example, see the `aux_LoadProgram` function presented earlier (page 8, specifically "step 4". Then, in function `main` (page 6), change the call to `aux_LoadProgram` to:

```
err = aux_LoadProgram(&amx, argv[1], NULL, srun_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 srun_Monitor(AMX *amx)
{
  switch (amx->dbgcode) {
  case DBG_INIT:
    return AMX_ERR_NONE;
  case DBG_LINE:
    /* check whether an "abort" was requested */
    return abortflagged ? AMX_ERR_EXIT : AMX_ERR_NONE;
  default:
    return AMX_ERR_DEBUG;
  } /* switch */
}
```

The debug hook *must* return `AMX_ERR_NONE` on the `DBG_INIT` event, otherwise it will receive no further events. The only other event captured by this particular debug hook function is `DBG_LINE`, which notifies the start of a statement on a new source code line. If the debug hook returns an error code other than `AMX_ERR_NONE` on the `DBG_LINE` event, the abstract machine aborts execution and returns that error code.

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.

• 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 maximum stack and heap usage. To this end, the example below extends the debug "monitor" function of the previous section, and adds another refinement at the same time.

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

  switch (amx->dbgcode) {
  case DBG_INIT:
    amx_Flags(amx,&flags);
    return (flags & AMX_FLAG_NOCHECKS) ? AMX_ERR_DEBUG : AMX_ERR_NONE;

  case DBG_LINE:
    /* 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;

  default:
    return AMX_ERR_DEBUG;
  } /* switch */
}
```

Appendix E covers the memory layout

This extended version of `amx_Monitor` still checks the `abortflagged` variable (which is set on a Ctrl-C or Ctrl-Break signal), but on the same `DBG_LINE` code 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);
```

In the new implementation of `amx_Monitor`, the handling of the `DBG_INIT` code has also changed, for an entirely different reason. The stack/heap monitoring and the program abort functionality (Ctrl-C or Ctrl-Break) depend on the regular arrival of `DBG_LINE` events. When a SMALL program is compiled without debug information, no line number information is present in the P-code and no `DBG_LINE` event will be send. 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 preformance. However, the AMX kernel will still call the debug hook for other events and, with our implementation of `amx_Monitor`, this events have no effect except that they take time.

When the debug hook returns an error code on the `DBG_INIT` event, it will not be called for any other events. Since our debug hook only "listens" to the `DBG_LINE` event and that event is non-functional when the P-code contains no line number information, we might as well return an error code on `DBG_INIT` in that situation. The upshot is that the overhead of the function call is avoided.

• Preparing for memory-hungry scripts

The core run-time files that build the abstract machine executive (`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 executive is made to be linked into host applications and,

* 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.

in practice, diverse host applications use dissimilar memory allocation schemes —from instrumented versions of `malloc` to garbage collection algorithms.

The drawback of this design, however, is that the address range that a compiled script runs in cannot easily grow: the executive 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 executive 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.

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

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").

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

  } __except (srun_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 page 6. 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 commited. 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. SMALL 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 srun_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 kBytes 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).

Writing extension modules: 23
Init/Cleanup functions: 24

Another change in `main` in comparison with the first implementation at page 6 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 SMALL 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 should be passed to `amx_Exec`. For numeric "value" parameters, this involves no more than adding the parameters at the end of the parameter list of `amx_Exec` and adjusting the fourth parameter ("`numparams`") to reflect the number of parameters. When the parameter is a reference parameter or an array, there is a complexity: these parameters are passed by address, but the abstract machine cannot access memory owned by the host. Instead, the host program must allocate memory in the address space of the abstract machine and copy the parameter into this memory block. 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:

```
err = amx_Allot(&amx, strlen(argv[3]) + 1, &amx_addr, &phys_addr);
if (err)
  ErrorExit(&amx, err);
amx_SetString(phys_addr, argv[3], 0);
```

The above snippet passes the string as an "unpacked" string, meaning that in the script, every cell holds one character. Therefore, the snippet allocates a number of cells equal to the string length, plus one for the zero terminator. The `amx_Allot` function returns two addresses; here stored in `amx_addr` and `phys_addr`. The `amx_addr` variable contains the memory address relative to the abstract machine —this is the address that must be passed to `amx_Exec`. The `phys_addr` variable is the address that the host program uses to store data into the abstract machine or to read the results. In this example, the host program simply calls `amx_SetString` to store the `argv[3]` string into the abstract machine's memory.

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.

See the Small booklet "The Language" for details on variable arguments

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_Allot` mentioned above, and it also shows how to pass one extra paramater through `amx_Exec`.

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

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

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

  err = srun_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_Allot(&amx, strlen(argv[3]) + 1, &amx_addr, &phys_addr);
  if (err)
    ErrorExit(&amx, err);
  amx_SetString(phys_addr, argv[3], 0);

  err = amx_Exec(&amx, NULL, index, 1, amx_addr);
  if (err)
    ErrorExit(&amx, err);

  amx_GetString(output, phys_addr);
  amx_Release(&amx, amx_addr);
  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_Allot`. A single call to `amx_Release` can undo multiple `amx_Allot`'s, see the function description at page 54.

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 SMALL booklet "The Language". The essential modification is the keyword `public` that is prefixed to the function name "`rot13`".

```
main()
    {
    printf("Please type the string to mangle: ")

    new str[100]
    getstring(str, sizeof str)
    rot13(str)

    printf("After mangling, the string is: ^"%s^"^n", str)
    }

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 "srun", we can execute the script with:

```
srun rot13.amx rot13 hello-world
```

Although the "ROT13" script contains a `main` function, it won't execute in this particular case.

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

1 allocate space for the array in the abstract machine with `amx_Allot`;
2 copy the array into the abstract machine using the "physical address" pointer that `amx_Allot` returned;
3 call the public function, passing the "AMX address" pointer as a parameter;
4 optionally copy the array back, out of the abstract machine —again using the "physical address" pointer;

5 free the memory block in the abstract machine with `amx_Release`.

The implementation of "srun" that calls the ROT13 script (page 18) 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 SMALL. 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 SMALL 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 18 for the context of this snippet):

```
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 amx_addr, *phys_addr;

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. allocate memory in the abstract machine */
  err = amx_Allot(&amx, Number, &amx_addr, &phys_addr);
  if (err == AMX_ERR_NONE) {

    /* 2. copy values into the abstract machine */
    memcpy(phys_addr, Values, Number * sizeof(cell));

    /* 3. call the public function with the "AMX address" */
    err = amx_Exec(&amx, (cell*)&Mean, index, 2, amx_addr, Number);
    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, amx_addr);

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

} /* if */
```

This example may appear a rather abstract twist of mind: "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 is 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`.

Extension modules

An extension module provides a SMALL program with application-specific ("native") functions. 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 SMALL programs.

• 1. Writing the native functions

Every native function must have the following prototype:

```
cell AMX_NATIVE_CALL func(AMX *amx, 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 reference, function `amx_GetAddr` converts the "abstract machine" address from the "`params`" array to a physical address. The pointer that `amx_GetAddr` returns lets you access variables inside the abstract machine directly. Function `amx_GetAddr` also verifies whether the input address is a valid address.

Strings, like other arrays, are always passed by reference. However, neither packed strings nor unpacked strings are universally compatible with C strings (on Big Endian computers, packed strings are compatible with C strings). Therefore, the abstract machine API provides two functions to convert C strings to and from SMALL strings: `amx_GetString` and `amx_SetString`.

See page 102 for the memory layout of arrays and page 29 for an example

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, cell *params)
{
  /* power(value, exponent);
   *   params[1] = value
   *   params[2] = exponent
   */
  cell result = 1;
  while (params[2]-- > 0)
    result *= params[1];
  return result;
}

static cell n_sqroot(AMX *amx, 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);
```

Example program that calls amx_Register: 6

The first example of "host applications" for the SMALL 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 SMALL 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 33; 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 SMALL abstract machine. It is common practice, however, that instead of writing *new* functions for SMALL, you will make a set of existing C/C++ functions available to SMALL. To "glue" the existing functions to SMALL, 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++–SMALL boundaries, plus that they provide templates for writing any kind of native functions.

• Pass-by-value, the simplest case

The SMALL 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, 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 SMALL 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 SMALL 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, 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`" 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.

• Strings

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

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

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

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

Its wrapper function is:

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

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

When the SMALL abstract machine passes an array to a native function, it passes the base address of the array. This address, however, is relative to the data section of the abstract machine; it is not a pointer that the native function (in C/C^{++}) can use as is. The function `amx_GetAddr` translates an "abstract machine address" (in `params[1]` in the above example) to a physical pointer for the host application (i.e. `cstr`).

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`. 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, 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: SMALL 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, cell *params)
{
  time_t r;
  cell *cptr;

  assert(sizeof(cell) == sizeof(time_t));
  amx_GetAddr(amx, params[1], &cptr);
  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_GetAddr`.

• 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 returned by `amx_GetAddr` now points to the first cell of potentionally many cells.

Memory lay-out of arrays: 102

Multi-dimensional arrays must be handled differently, though, as the memory lay-out differs between C/C++ and SMALL. In comparison with C/C++, two-dimensional arrays in SMALL 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 SMALL'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 `amx_GetAddr`.

For an example, I use the OpenGL function `glMultMatrixf` which multiplies a given 4×4 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, cell *params)
{
  cell *cptr;

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

For this example, I selected the OpenGL maxtrix muiltiplication 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 SMALL 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 finctions, 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: 12

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

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

  amx_StrParam(amx, params[1], pstr);
  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_LogFileExit(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 28), 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 SMALL 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 SMALL compiler. The SMALL compiler uses this `#pragma` to record which extension modules are actually referred to from the script. The SMALL 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 page 24 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 page 26.

◇ 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 page 78.

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 SMALL with those for other scripting languages, you may remark that the wrapper functions for SMALL are relatively small and easy. Notably, SMALL 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 SMALL script passes in. The wrapper function *could have* check this number of arguments, because SMALL passes the number of bytes that the native function receives in `params[0]`, but in most cases this extra checking is redundant.

isalpha() wrapper: 26

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, SMALL requires the definitions of the native functions (in SMALL syntax), in addition to the implementation —this was the third step in the list at the start of the chapter "Extension modules" (page 23).

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:

glMultMatrixf() wrapper: 30

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

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

Parameters checks that you may want to do at run-time, for the sake of security, are the validiy of addresses that you receive. For every reference parameter or array, your native function calls `amx_GetAddr` to convert an address relative to the abstract machine to a pointer usable by C/C^{++}. As SMALL does not allow

the script programmer to freely manipulate pointers, the addresses that a native function receives are *under normal circumstances* always valid, but a modified version of the SMALL compiler (or perhaps bugs in the compiler and/or abstract machine) may possibly be exploited to pass invalid addresses to a native function.

If security is important for your product, you should check the return value of `amx_GetAddr`; the function returns `AMX_ERR_MEMACCESS` if the input pointer is invalid. When using the macro `amx_StrParam`, the pointer to the allocated memory is set to `NULL` if the address of the input pointer is invalid.

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 SMALL 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. For real CPU's, the equivalent of a "SYSREQ.C" would be a software-invoked interrupt.

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, 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 conjunction with `amx_Register`, which looks up a function from the "native function table" in the header of the compiled program file and stores the physycal 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.

On the other hand, there are situations where the set of native functions that are available to a SMALL program are fixed and known in advance —for example, for abstract machines embedded in (small) hardware devices. 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:

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

The difference with the declarations on page 26 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 SMALL 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 SMALL compiler omits it from the native function table. That is, this scheme creates more compact binary files.

* Turn to appendix E for details on the opcodes.

The default native function dispatcher cannot handle native function indices, so you must replace it 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:

```
int my_callback(AMX *amx, cell index, cell *result, 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;
}
```

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".

See page 60 for the defined error codes.

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 SMALL Abstract Machine eXecutive 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_Align16`, `amx_Align32`, `amx_Align64`

amx_Allot Reserve heap space in the abstract machine

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

`amx` The abstract machine.

`cells` The number of cells to reserve.

`amx_addr` The address of the allocated cell as the SMALL program (that runs in the abstract machine) can access it.

`phys_addr` The address of the cell for C programs to access.

Notes: The intended purpose for `amx_Allot` and `amx_Release` is to pass arrays and reference arguments to public functions as parameters. A SMALL function can only access memory inside its abstract machine. If a parameter is to be passed "by reference" to a SMALL 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. An added complexity is that the abstract machine uses addresses that are relative to the data section of the abstract machine, and the host program uses address relative to the environment that the operating system gives it.

`amx_Allot` allocates memory cells inside the abstract machine and it returns *two* addresses. The `amx_addr` parameter is the address of the variable relative to the "data section" of the abstract machine; this is the value you should pass to `amx_Exec`. Parameter `phys_addr` holds the address relative to the host program's address space. So a C program can use this address and write into the allocated memory.

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

See also: `amx_Exec`, `amx_Release`

amx_Callback The default callback

Syntax: `int amx_Callback(AMX *amx, cell index, cell *result, 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.

See page 60 for the defined error codes.

Notes: The abstract machine has a default callback function, which works in conjunction 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_Clone Clone an abstract machine

Syntax: `int amx_Clone(AMX *amxClone, AMX *amxSource, void *data)`

`amxClone` The new abstract machine. This variable is initialized with the settings of the `amxSource` abstract machine. Before calling this function, all fields of the `amxClone` structure variable should be set to zero.

`amxSource` The abstract machine whose code is to be shared with the cloned abstract machine and whose data must be copied. This abstract machine has to be initialized (with `amx_Init`).

`data` The memory block for the cloned abstract machine. This block must hold the static (global) data, the stack and the heap.

Notes: Use `amx_MemInfo` to query the size of the static data and the stack/heap of the source abstract machine. The memory block to allocate for the `data` parameter should have a size that is the sum of the global data and the stack/heap size.

The cloned abstract machine has a separate data section and a separate stack, but it shares the executable code with the source abstract machine. The source abstract machine should not be deleted while any cloned abstract machines might still be active.

The state of the data section (the global and static variables) are copied from the source abstract machine to the clone at the time that `amx_Clone` is called. If the source abstract machine has modified any global/static variables before it is cloned, the clone will have these values as its initial state. In practice, it may be advisable *not* to "run" the source abstract machine at all, but to use it only for cloning and run the clones.

See also: `amx_Init`, `amx_MemInfo`

---

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 Small 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_Debug The default debug hook

Syntax: `int amx_Debug(AMX *amx)`

`amx` The abstract machine.

Returns: The debug hook should return an error code, or `AMX_NO_ERROR` for no error.

Notes: The default debug function is a stub that immediately returns. Programs can replace the default debug hook function to monitor symbols and to trace through code step by step. The debugger interface is described in a separate document and through an example program in the distribution: `SDBG.C`.

See also: `amx_SetDebugHook`

amx_Exec Run code

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

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

`retval` Will hold the return value of the called function upon return.

`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.

`numparams` The number of function parameters that follow.

`...` Optional parameters for the function. All these parameters must be cast to the type `cell`, which is usually a 32-bit integer.

Notes: This function 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`.

In situations where dealing with variable arguments is inconvenient, use `amx_Execv`.

See also: `amx_Execv`, `amx_FindPublic`, `amx_Register`

---

amx_Execv Run code

Syntax:
```
int amx_Exec(AMX *amx, cell *retval, int index, int
             numparams, cell params[])
```

`amx`	The abstract machine from which to call a function.
`retval`	Will hold the return value of the called function upon return.
`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.
`numparams`	The number of function parameters that follow.
`params`	An array with the parameters for the function.

Notes: This function calls the callback function for any native function call that the code in the AMX makes. `amx_Execv` assumes that all native functions are correctly initialized with `amx_Register`.

See also: `amx_Exec`, `amx_FindPublic`, `amx_Register`

---

amx_FindNative Return the index of a native function

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

`amx`	The abstract machine.
`funcname`	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 *funcname, int *index)`

`amx` The abstract machine.

`funcname` 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 *varname,
                   cell *amx_addr)
```

`amx` The abstract machine.

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

`amx_addr` Upon return, this parameter holds the variable address relative to the abstract machine.

Notes: The returned address is the address relative to the "data section" in the abstract machine. Use `amx_GetAddr` to acquire a pointer to its "physical" address.

See also: `amx_FindPublic`, `amx_GetAddr`, `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_CHAR16` if a character is 16-bits rather than the default of 8 bits

`AMX_FLAG_DEBUG`	if the program carries symbolic information
`AMX_FLAG_COMPACT`	if the program is stored in "compact encoding"
`AMX_FLAG_BIGENDIAN`	if multi-byte values are stored in "Big Endian" order
`AMX_FLAG_NOCHECKS`	if the compiled P-code does not include line number information and run-time (bounds) checks

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 line number information, 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 SMALL 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_GetAddr Resolve an AMX address

Syntax: `int amx_GetAddr(AMX *amx,cell amx_addr,cell **phys_addr)`

`amx` The abstract machine.

`amx_addr` The address relative to the abstract machine.

`phys_addr` A pointer to the variable that will hold the memory address of the indicated cell. If the `amx_addr` parameter is not a valid address inside the abstract machine, `phys_addr` will be set to `NULL`.

Notes: This function returns the memory address of an address in the abstract machine. One typically uses this function in an extension module, because it allows you to access variables inside the abstract machine.

---

amx_GetNative Return a native function name

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

`amx` The abstract machine.

`index` The index of the requested function. Use zero to retrieve the name of the first native function.

`funcname` The string that will hold the name of the native function.

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 *funcname)`

`amx` The abstract machine.

`index` The index of the requested function. Use zero to retrieve the name of the first public function.

`funcname` The string that will hold the name of the public function.

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_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 *varname,
                  cell *amx_addr)
```

`amx` The abstract machine.

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

`varname` The string that will hold the name of the public variable.

`amx_addr` Upon return, this parameter holds the variable address relative to the abstract machine.

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

The returned address is the address relative to the "data section" in the abstract machine. Use `amx_GetAddr` to acquire a pointer to its "physical" address.

See also: `amx_FindPubVar`, `amx_GetAddr`, `amx_GetPublic`, `amx_NameLength`, `amx_NumPubVars`

---

amx_GetString Retrieve a string from the abstract machine

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

`dest` A pointer to a character array of sufficient size to hold the converted source string.

`source` A pointer to the source string. Use `amx_GetAddr` to convert a string address in the AMX to the physical address.

Notes: This function converts both packed strings and unpacked strings from the "SMALL" format to the "C" format.

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.

`program` A pointer to the bytecode stream of the program.

Notes: `amx_Init` initializes the abstract machine with the settings from the binary file. Before calling this function, you should set the `amx` structure variable to all zeros.

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: `code_size` and `reloc_size`. Both fields are sizes of buffers that must be allocated for `amx_InitJIT`. The abstract machine will be compiled into the block `native_code`, which must have the size `code_size` (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) always too large. Function `amx_InitJIT` adjusts the `code_size` 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 futhermore have the appropriate priveledges to execute machine instructions. See page 73 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 E 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.

See also: `amx_Clone`

---

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 SMALL 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_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()` returns.

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 function 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 function returns the error code `AMX_ERR_NOTFOUND`, one or more native functions that are used by the SMALL program are not found in the provided list. You can call `amx_Register` again to register additional 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 amx_addr)`

`amx` The abstract machine.

`amx_addr` The address of the allocated cell as the SMALL program (that runs in the abstract machine) sees it. This value is returned by `amx_Allot`.

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 `amx_addr`. That is, a single call to `amx_Release` can free multiple calls to `amx_Allot` if you pass the `amx_addr` value of the first allocation.

See also: `amx_Exec`, `amx_Release`

---

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 conjunction 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. See `amx_Debug` for the prototype and calling convention of a debug hook routine.

Notes: If you use a non-default debug hook routine, you should set it before calling `amx_Init`.

To set the default debug routine, set parameter `debug` to the function `amx_Debug`.

---

amx_SetString Store a string in the abstract machine

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

`dest` A pointer to a character array in the AMX where the converted string is stored. Use `amx_GetAddr` to convert a string address in the AMX to the physical address.

`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.

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(cell *cstring, int *length)`

`cstring` The string in the abstract machine.

`length` This parameter will hold the string length upon return.

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).

See also: `amx_GetAddr`, `amx_GetString`, `amx_SetString`, `amx_StrParam`

---

amx_StrParam Get a string parameter from an abstract machine

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

`amx`	The abstract machine.
`param`	The parameter number.
`result`	A variable that will hold the result on return.

Notes: This **macro** allocates a block of memory (with `alloca`) and copies a string parameter (to a native function) in that block. See page 28 for an example of using this macro.

See also: `amx_GetAddr`, `amx_GetString`, `amx_StrLen`

---

amx_UTF8Check Check whether a string is valid UTF-8

Syntax: `int amx_UTF8Check(const char *string)`

`string`	A zero-terminated string.

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).

See also: `amx_UTF8Get`, `amx_UTF8Put`

---

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_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 requried 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`

---

aux_StrError Get a text description of an error

Syntax: `char *aux_StrError(int errnum)`

`errnum` The error number.

Notes: This function returns a pointer to a static string with a description of the error number `errnum`.

Error numbers: 60

A few "error" codes, like `AMX_ERR_SLEEP`, do not really denote an error situation. For those error codes and for invalid values of `errnum`, the function returns a description that is enclosed in parentheses.

Error codes

`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_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.

APPENDIX A

Building the compiler

The C sources of the compiler 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 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.

There are two makefiles and a set of project files for building the SMALL compiler. You can choose the one that is most convenient:

- ◇ For users of Microsoft Visual C/C++, "project" and "workspace" files for the compiler and abstract machine that can be found in the "msvc" subdirectory of where SMALL is installed.
- ◇ An alternative, multi-platform tool for building SMALL is "CMake"; see also appendix C. A project file for CMake is included in the directory where SMALL is installed or in the the "source" subdirectory (depending on the installation package).
- ◇ CMake creates project files or makefiles for a few selected compilers and platforms. However, a pre-build "`makefile.win`", for DOS/Windows, multiple compilers and Borland/Opus MAKE is included in a "compiler" subdirectory.
- ◇ "`makefile.linux`", for Linux using GCC and GNU MAKE; and a "`makefile.freebsd`" for FreeBSD.

All of the above project files build the compiler as a console-mode executable. To build the compiler as a library (statically linked or a shared library/DLL), see page 65.

The "`makefile.win`" is for DOS/Windows and supports multiple compilers. I tested this makefile with Borland MAKE and Opus MAKE. To select a compiler, you have to define a constant on the command line for the MAKE utility; for example, when using Opus MAKE and the Microsoft Visual C/C++ compiler, you would say:

```
make -fmakefile.win MSC=1
```

Alternatively, you can modify the makefile and comment out the compiler definitions that you do not want. If you furthermore rename `makefile.win` to simply `makefile`, you can build the compiler with the simple command:

```
make
```

The makefile for Linux is "`makefile.linux`". 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 SMALL: 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. the GCC compiler) are sensitive to the way that lines are ended. The utility "`dos2unix`" is the most convenient way to translate source files. That behind you, you should be able to say:

```
make -f makefile.linux
```

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.

CMake is a tool that *builds* makefiles or IDE project files from a general purpose project description file and a set of pre-build compiler definition files. As an alternative to use the included makefiles, you can therefore use CMake to build a dedicated makefile for your platform (Windows/Unix) and your compiler.

The CMake configuration files are set up so that you launch the CMake utility from the "parent" directory of the subdirectories where the source files for the compiler or the abstract machine reside. CMake will generate the makefiles in this parent directory and `make` will build the executable program there too.

• 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 SMALL compiler that outputs 16-bit P-code, add the definition `BIT16` 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 comforms 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 there is an "`#if defined(BIT16)`" around some code, because I use the constants `SHORT_MAX` and `LONG_MAX` from `LIMITS.H` as well.
N.B. The SMALL tools are not regularly tested with the `BIT16` definition.

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 SMALL 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 kBytes 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 SMALL compiler includes a preprocessor that does text substitutions (with or without parameters). The text matching capabilities of the SMALL 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 SMALL 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 `NO_UTF8`. The UTF-8 decoder in SMALL supports the full 31-bit UCS-4 character set.

A few functions of the SMALL compiler are non-essential gadgets. In cases where the size of the compiler counts, these can be removed by compiling with the `SC_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 `SC.CFG` file, whose options are implicitly read.

• Summary of definitions

`AMX_COMPACTMARGIN`	The size of the buffer needed for the "compact encoded" file format. See page 96 for details on compact encoding. The default value is 64 (cells). When this value is set to zero, support for compact encoding is removed alltogether from the abstract machine. When support for compact encoding is desired, it is advised to set this value to at least 30.
`BIT16`	cell is 16-bits, rather than 32-bits

`LINUX`	compile for Linux (or perhaps other Unix versions)
`NDEBUG`	compile without assertions
`NO_MAIN`	remove main() and I/O functions from the program
`NO_DEFINE`	remove the text preprocessor from the SMALL compiler (i.e. the `#define` directive)
`NO_UTF8`	remove the UTF-8 reading capability
`SC_LIGHT`	remove cross-reference and response file support

• Embedding the compiler into an application

When you want to link the SMALL 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 polution of the global namespace by the many, many functions and global variables of the SMALL compiler.

◇ Overrule the functions that the SMALL compiler calls for input/output.

The archive contains the file `LIBSC.C` which illustrates how to perform these steps. Basically, you implement all file I/O functions that the SMALL 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 `sc_compile`, passing in all arguments. The prototype of the function is:

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

As you can see, this function looks like the standard function `main`; when calling `sc_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 `sc_compile`) are `sc_addconstant` and `sc_addtag`. Function `sc_compile` removes all symbols before returning, including all constants and tagnames that you added with `sc_addconstant` and `sc_addtag`.

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

An example command line to build `LIBSC` as a DLL is:

LIBSC.DLL, with Borland C^{++} version 5.x

```
bcc32 -tWD -DSC_DLL libsc.c
```

This creates the DLL only; you may want to run IMPLIB *to generated the associated import library.*

The `LIBSC.C` file is written in a portable way and it can be compiled with other compilers with comparative ease. Compiling `LIBSC.C` to a "shared library" (Linux, UNIX) is not much different than the compiling to a DLL, but you may have to adjust the source code (to strip out the `main` function, for example).

Note that the exported functions in `LIBSC` 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 SMALL 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 `SCLIB` can be driven from `RUNDLL/RUNDLL32`. The command line to use, for a 32-bit version, is:

`rundll32 libsc.dll,Compile` *options* `hello.sma` 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 succesfully built __stdcall functions that use variable length argument lists.

APPENDIX B

Building the Abstract Machine eXecutive

Project files to build the example "`srun`" 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 eXecutive" (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. 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;
- GNU GCC optimized: still implemented in C, but using specific GNU GCC extensions that make it significantly faster than the ANSI C version;
- Intel Pentium assembler: this is a single design, but doubly implemented to support a wide range of assemblers;
- Just-In-Time compilers: the fastest core (but the least portable).

Next to the basic AMX library, the toolkit comes with various extension modules (native function libaries) 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 eXecutive".

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 "default" build for the tools is for DOS/Windows. To compile for Linux, add the macro definition `LINUX` on the compiler's command line.

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 `NOPROPLIST`.

sidxWin32 Console 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 page 76 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 SMALL compiler is flexible in its handling of codepages and translation of extented ASCII and UTF-8 to wide characters (i.e., Unicode). For the host application, there are essentially two approaches:

See also page 76 for terminals supporting Unicode or UTF-8

Support Unicode or UCS-4 and interpret unpacked strings as strings holding "wide" characters. The SMALL 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 SMALL compiler into a surrogare pair.

1 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. 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.

Calling conventions are always an important issue in porting software. The SMALL 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 eXecutive (e.g. `amx_Init`), includ-

ing 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. See page 78 for details.

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.

• Summary of definitions

`AMX_COMPACTMARGIN` The size of the buffer needed for the "compact encoded" file format. See page 96 for details on compact encoding. The default value is 64 (cells). When this value is set to zero, support for compact encoding is removed alltogether from the abstract machine. When support for compact encoding is desired, it is advised to set this value to at least 30.

`AMX_NATIVE_CALL` calling convention of native functions (applies to `AMX.C` and to extension modules);

`AMX_NODYNALOAD` disable support for dynamically loadable extension modules, see the discussion at page 33 (`AMX.C`);

`AMX_TERMINAL` for `AMXCONS.C`, do not use console functions (Win32 console, ANSI/VT100 or plain console);

`AMXAPI` calling convention of interface functions; this overrides any `CDECL` or `STDECL` macros (`AMX.C`);

`AMXEXPORT` calling convention of initialization and clean-up functions of extension modules;

`ASM32` compile the assembler version (`AMX.C`);

`CDECL` sets `AMXAPI` to `__cdecl`, for compatibility with the assembler core (`AMX.C`);

`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
`JIT`	add support for the Just-In-Time compiler (`AMX.C`);
`LINUX`	compile for Linux (or perhaps other Unix versions);
`NDEBUG`	compile without assertions (all files);
`NOPROPLIST`	remove the get/set property functions from `AMXCORE.C`;
`STDECL`	sets `AMXAPI` to `__stdcall`, for compatibility with the assembler core (`AMX.C`);
`UNICODE, _UNICODE`	Enable Unicode in the console I/O module and possibly other auxiliary libaries.
`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 `SRUN`, a simple P-code interpretor that is developed starting at page 6.

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 SMALL is installed.

- **ANSI C (see the GNU C section for Linux)**

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

```
bcc srun.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 srun.c amx.c amxcons.c amxcore.c
```

Microsoft Visual C/C^{++} version 5.0 or 6.0, 32-bit

```
cl srun.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 /l=nt srun.c amx.c amxcore.c amxcons.c
```

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

• Assembler core for the Abstract Machine eXecutive

Marc Peter's assembler implementation of the Abstract Machine eXecutive currently runs with all 32-bit C compilers for Microsoft Windows. It is (approximately) five times faster than the ANSI C version. As you can see on the command line, the C files need the `ASM32` macro to be defined.

There are two "calling convention" issues in the assembler implementation (in addition to those mentioned at page 68):

- ◇ 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 Microsoft's MASM is now freely available from Microsoft's WEB site, and that the free "Netwide Assembler" is now also supported). 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 two pre-compiled assembler files were both build from the file `AMXEXEC.ASM` (but with different options). This assembler file is compatible with Microsoft MASM, Borland TASM and Watcom WASM. The Netwide Assembler (NASM)

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 is a free assembler which runs on a variety of platforms.

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.

I have had troubles with the incremental linker when mixing assembler with C/C++, for both Borland and Microsoft compilers. When the program fails for mysterious reasons, or when the debugger shows assembler code or variable addresses that clearly do not match the associated source code, first do a full build (and especially a full "link").

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

```
bcc32 -DASM32 -TdCDECL -TdSTACKARGS srun.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 on the TASM32.*

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

```
nasmw -f obj -d BORLAND amxexecn.asm
bcc32 -DASM32 srun.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 -f elf amxexecn.asm
gcc -o srun -DLINUX -DASM32 -I../linux srun.c amx.c amxcore.c (...)
(...) amxcons.c ../linux/getch.c amxexecn.o -ldl
```

Most Linux distributions use the "elf" file format. See page 75 for the extra file `getch.c` *and page 78 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 -DASM32 srun.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 -DASM32 srun.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 -DASM32 -DAMXAPI=__stdcall srun.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 /dASM32 srun.c amx.c amxcore.c amxcons.c amxexec.asm
```

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

• Just-In-Time compiler

The third option is to add the Just-In-Time compiler, plus support routines. The JIT compiles 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 eXecutive (which was pretty fast already). To add support for the JIT, you must define the macro "`JIT`" via a command line switch.

In addition to compiling with the `JIT` macro defined, the host application should call the `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 SMALL 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-compiler generates the instructions has the NX/XD bit *cleared*.

The JIT-compiler 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.C` 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 compiler requires write-access to its own code segment: the JIT-compiler 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 `srun_jit.c` which is a modification of the "SRUN" program (the example program for the embedding of the abstract machine, see page 6) for the JIT-compiler. This example program lays out the steps described above.

There are, in fact, three versions of the JIT:

`AMXJITR.ASM` uses register based calling conventions and requires Watcom C/C++;

`AMXJITS.ASM` uses `__cdecl` or `__stdcall` calling conventions (both are stack based) and should work with other Win32 compilers.

`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 (like support for some debug opcodes).

Note that the JIT (as it is today) does not support the debugger hooks.

Borland C++ version 5.02, 32-bit

```
bcc32 -DJIT -Tm2 srun_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 /dJIT srun_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 -f elf amxjitsn.asm
gcc -o srun -DLINUX -DJIT -I../linux srun_jit.c amx.c amxcore.c (...)
(...) amxcons.c ../linux/getch.c amxjitsn.o -ldl
```

Most Linux distributions use the "elf" file format. See page 75 for the extra file `getch.c` *and page 78 for the option* `-ldl` *which causes the inclusion of the library* `libdl`

• Direct threaded interpreter with GNU C extensions

The AMX.C file has special code for the GNU C compiler (GCC), which makes the Abstract Machine eXecutive about twice as fast as the ANSI C version. However, the assembler core and the JIT are faster still.

GNU GCC for DOS/Windows (DJGPP version 2.01), 32-bit

```
gcc -o srun.exe srun.c amx.c amxcore.c amxcons.c
```

When running with all warnings enabled, option "`-Wall`*", the GNU C compiler suggests extra parantheses around subexpressions in a logical expression; I do not agree with the desirability of extra parantheses in this case, so I have not added them.*

GNU GCC for Linux, FreeBSD and OpenBSD

```
gcc -o srun -DLINUX -I../linux srun.c amx.c amxcore.c amxcons.c (...)
(...) ../linux/getch.c -ldl
```

You must add the "`LINUX`*" option for alternative support code. 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 page 78). Dynamic linking requires the inclusion of the library* `libdl`*.*

• 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 fallback 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 SMALL 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 `termin` would be:

```
wcl386 /dAMX_TERMINAL /l=nt srun.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 `SRUN.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 artithmetic in your Small 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 Small 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 srun.c amx.c amxcore.c amxcons.c float.c
```

The original line read: `bcc srun.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 srun -DLINUX -DFLOATPOINT -I../linux srun.c amx.c amxcore.c (...)
(...) amxcons.c float.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 SMALL 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 dynamicall 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" and they must also have the "`__stdcall`" calling convention. For that purpose, `AMX.H` defines the macro `AMXEXPORT`: it is suggested that the definitions of `amx_`*Filename*`Init` and `amx_`*Filename*`Cleanup` are marked with this macro and that you set it to the appropriate (compiler-dependent) calling convention on the compiler command line.

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. Refer to page 68 for details (set all three macros `AMX_NATIVE_CALL`, `AMXAPI` and `AMXEXPORT` to `__stdcall`).

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="__stdcall _export" (...)
(...) -DAMX_NATIVE_CALL=__stdcall -DAMXAPI=__stdcall float.c amx.c float.rc
```

See also page 33 for the filename convention of dynamically loadable extension modules

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` —nobably `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 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_EXEC`	for `amx_Exec` and `amx_Execv`
`AMX_FLAGS`	for `amx_Flags`
`AMX_GETADDR`	for `amx_GetAddr`
`AMX_INIT`	for `amx_Init` and `amx_InitJIT`
`AMX_MEMINFO`	for `amx_MemInfo`
`AMX_NAMELENGTH`	for `amx_NameLength`

AMX_NATIVEINFO	for amx_NativeInfo
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

APPENDIX C

Using CMake

CMake is a cross-platform, open-source make system, which generates "`makefile`'s" or project files for diverse compilers and platforms. You can find more information on CMake plus a freely downloadable copy on http://www.cmake.org/.

The SMALL toolkit comes with a CMake project file that builds the compiler, a simple run-time program that embeds the abstract machine, and a simple console debugger. The CMake project file is in the "source" subdirectory of where the SMALL toolkit is installed, when you installed the self-extracting setup for Microsoft Windows. When unpacking the SMALL source code from a `.ZIP` or `.TGZ` archive, the CMake project file is in the main directory where you unpacked the archive into.

- **Microsoft Windows**

1. Launch `CMakeSetup`.
2. Select for the source code directory, the "source" subdirectory in the directory tree for the toolkit.

   For example, if you installed the toolkit in `C:\Small`, the source directory is `C:\Small\source`.
3. Select as destination the "source" subdirectory; the destination and source directories are thus the same.
4. Select the compiler to use, as well.
5. Click on the "Configure" button. After an initial configuration, you may have items displayed in red. These may need adjustment, for instance the "`EXECUTABLE_OUTPUT_PATH`" item. After adjustment, you click "Configure" once more.
6. Click on the "OK" button. This exits the `CMakeSetup` program after creating a number of files in the "source" subdirectory.
7. Build the program in the usual way. For Microsoft Visual C/C++, CMake has created a Visual Studio project and "Workspace" file; for other compilers CMake builds a `makefile`.

• **Linux / Unix**

1. Change to the directory where the archive was extracted into. For example, if you unpacked the toolkit in `/usr/Small`, go to that directory.
2. Launch "`ccmake .`" (the "`.`" stands for the current directory).
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. These may need adjustment, for instance the "`EXECUTABLE_OUTPUT_PATH`" item. After adjustment, you type "c" once more.
4. Press the "g" button for "generate and quit". Then build the program by typing "make".

APPENDIX D

Design of the Abstract Machine eXecutive

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.

There is only one real reason to use an abstract machine: cross-platform compatibility of the compiled binary code. At the time that SMALL was designed, both 16-bit and 32-bit platforms on the 80x86 processor series were important for me. By the time I can forget about 16-bit operating systems, alternate microprocessors (like PowerPC and DEC Alpha) may have become essential.

Other reasons (while not essential) are:

- ◇ 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 multithreading, 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 bytecode 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 SMALL language does not have the ability to execute bytecode from an array, but the abstract machine is not too tightly coupled to the language. That is, future versions of the SMALL 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
- ◇ 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 bytecode, 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 bytecode instruction (more on this later). For six bytecode 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 bytecode 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 SMALL "AMX", a two-register abstract machine (AMX stands for "Abstract Machine eXecutive"):

```
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 bytecode instructions (2 native instructions per bytecode 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 bytecode 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 SMALL 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 bytecode 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 an indirect 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 SMALL abstract machine is designed such that both indirect 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 SMALL system are in appendix E. 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`, see appendix E) 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`.

APPENDIX E

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 eXecutive 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

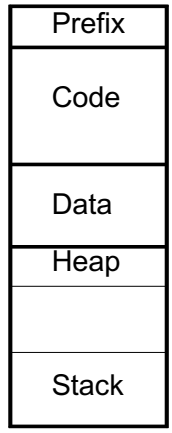


FIGURE 1: *Memory layout of the abstract machine*

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 startup information, and the definitions of native and public functions.

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 either in Little Endian or in compact encoding —see page 96 for details on compact encoding.

`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 7
`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 variables" table
`nametable`	4 bytes	offset to the symbol name table (file version 7)
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)
name table	variable	the symbol name table (file version 7; 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` *reserved* —currently unused
- `1` (`AMX_FLAG_DEBUG`) if set, the file contains symbolic (debug) information
- `2` (`AMX_FLAG_COMPACT`) if set, the file is compressed with "compact encoding" —see page 96
- `3` *reserved* —currently unused
- `4` (`AMX_FLAG_NOCHECKS`) if set, the code has no debug support at all (no array bounds-checking, no assertions, no line-tracing support)
- `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.

In versions 0 to 6 of the P-code files, the records in the public functions table have the format:

`address`	*cell size*	the address (relative to `COD`) of the function
`name`	*defsize - cell size*	the name of the public function

As is apparent, the name of the public function is present in the record. The maximum length of a name of a public function is limited to the size of the record (minus the number of bytes in a cell, for the bytes taken by the `address` field).

The format of the native functions table is very similar (see below —this is, again, the format for file versions 0–6). The order of the records in the table is important, because the parameter of the `SYSREQ.C` instruction is an index into the native functions table.

`address`	*cell size*	used internally, should be zero in the file
`name`	*defsize - cell size*	the name of the native function

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 "`name`" field holds the library name.

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.

As of file version 7, the compiled file includes a "name table". This table 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.

As the symbol names no longer need to be stored in the tables for the public and native functions, the public variables, the tags and the libraries, the records for these tables have changed too. Instead of a `name` field, the records contain a 4-byte offset, relative to the start of the file "prefix", to the start of the symbol

name in the name table. The record size in the header, "`defsize`", is set to the size of one cell plus the 4-byte offset —i.e. 8 for a 32-bit cell implementation and 12 for a 64-bit cell implementation. Below is the definition for a native/public function/variable in file format 7.

`address`	*cell size*	see descriptions for native/public functions/var.'s	
`nameofs`	4 bytes	offset to the symbol name, relative to prefix	

• 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 SMALL 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`.

opcode	mnemonic	parameters	semantics
1/2	LOAD.pri/alt	address	PRI/ALT = [address]
3/4	LOAD.S.pri/alt	offset	PRI/ALT = [FRM + offset]
5/6	LREF.pri/alt	address	PRI/ALT = [[address]]
7/8	LREF.S.pri/alt	offset	PRI/ALT = [[FRM + offset]]
9	LOAD.I		PRI = [PRI] (full cell)
10	LODB.I	number	PRI = "number" bytes from [PRI] (read 1/2/4 bytes)

11/12	CONST.pri/alt	value	PRI/ALT = value
13/14	ADDR.pri/alt	offset	PRI/ALT = FRM + offset
15/16	STOR.pri/alt	address	[address] = PRI/ALT
17/18	STOR.S.pri/alt	offset	[FRM + offset] = PRI/ALT
19/20	SREF.pri/alt	address	[[address]] = PRI/ALT
21/22	SREF.S.pri/alt	offset	[[FRM + offset]] = PRI/ALT
23	STOR.I		[ALT] = PRI (full cell)
24	STRB.I	number	"number" bytes at [ALT] = PRI (write 1/2/4 bytes)
25	LIDX		PRI = [ALT + (PRI × *cell size*)]
26	LIDX.B	shift	PRI = [ALT + (PRI << *shift*)]
27	IDXADDR		PRI = ALT + (PRI × *cell size*) (calculate indexed address)
28	IDXADDR.B	shift	PRI = ALT + (PRI << *shift*) (calculate indexed address)
29/30	ALIGN.pri/alt	number	Little Endian: PRI/ALT ^= *cell size*− number
31	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)
32	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
33/34	MOVE.pri/alt		PRI=ALT / ALT=PRI
35	XCHG		Exchange PRI and ALT
36/37	PUSH.pri/alt		[STK] = PRI/ALT, STK = STK − *cell size*
38	PUSH.R	value	Repeat *value* ×: [STK] = PRI, STK = STK − *cell size*
39	PUSH.C	value	[STK] = value, STK = STK − *cell size*
40	PUSH	address	[STK] = [address], STK = STK − *cell size*
41	PUSH.S	offset	[STK] = [FRM + offset], STK = STK − *cell size*
42/43	POP.pri/alt		STK = STK + *cell size*, PRI/ALT = [STK]
44	STACK	value	ALT = STK, STK = STK + value
45	HEAP	value	ALT = HEA, HEA = HEA + value
46	PROC		[STK] = FRM, STK = STK − *cell size*, FRM = STK
47	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.
48	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.
49	CALL	address	[STK] = CIP + 5, STK = STK − *cell size* CIP = address The CALL instruction jumps to an address after storing the address of the next sequential instruction on the stack.
50	CALL.pri		[STK] = CIP + 1, STK = STK − *cell size* CIP = PRI jumps to the address in PRI after storing the address of the next sequential instruction on the stack.
51	JUMP	address	CIP = address (jump to the address)
52	JREL	offset	CIP = CIP + offset (jump "offset" bytes from current position)
53	JZER	address	if PRI == 0 then CIP = [CIP + 1]
54	JNZ	address	if PRI != 0 then CIP = [CIP + 1]
55	JEQ	address	if PRI == ALT then CIP = [CIP + 1]
56	JNEQ	address	if PRI != ALT then CIP = [CIP + 1]
57	JLESS	address	if PRI < ALT then CIP = [CIP + 1] (unsigned)
58	JLEQ	address	if PRI <= ALT then CIP = [CIP + 1] (unsigned)
59	JGRTR	address	if PRI > ALT then CIP = [CIP + 1] (unsigned)
60	JGEQ	address	if PRI >= ALT then CIP = [CIP + 1] (unsigned)
61	JSLESS	address	if PRI < ALT then CIP = [CIP + 1] (signed)
62	JSLEQ	address	if PRI <= ALT then CIP = [CIP + 1] (signed)
63	JSGRTR	address	if PRI > ALT then CIP = [CIP + 1] (signed)
64	JSGEQ	address	if PRI >= ALT then CIP = [CIP + 1] (signed)
65	SHL		PRI = PRI << ALT
66	SHR		PRI = PRI >> ALT (without sign extension)
67	SSHR		PRI = PRI >> ALT with sign extension
68	SHL.C.pri	value	PRI = PRI << value
69	SHL.C.alt	value	ALT = ALT << value
70	SHR.C.pri	value	PRI = PRI >> value (without sign extension)
71	SHR.C.alt	value	ALT = ALT >> value (without sign extension)
72	SMUL		PRI = PRI * ALT (signed multiply)
73	SDIV		PRI = PRI / ALT (signed divide), ALT = PRI mod ALT
74	SDIV.alt		PRI = ALT / PRI (signed divide), ALT = ALT mod PRI
75	UMUL		PRI = PRI * ALT (unsigned multiply)
76	UDIV		PRI = PRI / ALT (unsigned divide), ALT = PRI mod ALT
77	UDIV.alt		PRI = ALT / PRI (unsigned divide), ALT = ALT mod PRI

78	ADD		PRI = PRI + ALT
79	SUB		PRI = PRI − ALT
80	SUB.alt		PRI = ALT − PRI
81	AND		PRI = PRI & ALT
82	OR		PRI = PRI | ALT
83	XOR		PRI = PRI ^ ALT
84	NOT		PRI = !PRI
85	NEG		PRI = −PRI
86	INVERT		PRI = ~PRI
87	ADD.C	value	PRI = PRI + value
88	SMUL.C	value	PRI = PRI * value
89/90	ZERO.pri/alt		PRI/ALT = 0
91	ZERO	address	[address] = 0
92	ZERO.S	offset	[FRM + offset] = 0
93/94	SIGN.pri/alt		sign extent the byte in PRI or ALT to a cell
95	EQ		PRI = PRI == ALT ? 1 : 0
96	NEQ		PRI = PRI != ALT ? 1 : 0
97	LESS		PRI = PRI < ALT ? 1 : 0 (unsigned)
98	LEQ		PRI = PRI <= ALT ? 1 : 0 (unsigned)
99	GRTR		PRI = PRI > ALT ? 1 : 0 (unsigned)
100	GEQ		PRI = PRI >= ALT ? 1 : 0 (unsigned)
101	SLESS		PRI = PRI < ALT ? 1 : 0 (signed)
102	SLEQ		PRI = PRI <= ALT ? 1 : 0 (signed)
103	SGRTR		PRI = PRI > ALT ? 1 : 0 (signed)
104	SGEQ		PRI = PRI >= ALT ? 1 : 0 (signed)
105	EQ.C.pri	value	PRI = PRI == value ? 1 : 0
106	EQ.C.alt	value	PRI = ALT == value ? 1 : 0
107/108	INC.pri/alt		PRI = PRI + 1 / ALT = ALT + 1
109	INC	address	[address] = [address] + 1
110	INC.S	offset	[FRM + offset] = [FRM + offset] + 1
111	INC.I		[PRI] = [PRI] + 1
112/113	DEC.pri/alt		PRI = PRI − 1 / ALT = ALT − 1
114	DEC	address	[address] = [address] − 1
115	DEC.S	offset	[FRM + offset] = [FRM + offset] − 1
116	DEC.I		[PRI] = [PRI] − 1
117	MOVS	number	Copy memory from [PRI] to [ALT]. The parameter specifies the number of bytes. The blocks should not overlap.

118	CMPS	number	Compare memory blocks at [PRI] and [ALT]. The parameter specifies the number of bytes. The blocks should not overlap.
119	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.
120	HALT	0	Abort execution (exit value in PRI), parameters other than 0 have a special meaning.
121	BOUNDS	value	Abort execution if PRI $>$ value or if PRI $<$ 0
122	SYSREQ.pri		call system service, service number in PRI
123	SYSREQ.C	value	call system service
124	FILE	size ord name	source file information pair: name and ordinal (see below)
125	LINE	line ord	source line number and file ordinal (see below)
126	SYMBOL	size offset flag name	symbol information (see below)
127	SRANGE	level size	symbol range and dimensions (see below)
128	JUMP.pri		CIP = PRI (indirect jump)
129	SWITCH	address	Compare PRI to the values in the case table (whose address is passed) and jump to the associated address.
130	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.
131/132	SWAP.pri/alt		[STK] = PRI/ALT and PRI/ALT = [STK]
133	PUSHADDR	offset	[STK] = FRM + offset, STK = STK − *cell size*
134	NOP		no-operation, for code alignment
135	SYSREQ.D	address	call system service directly (by address)
136	SYMTAG	value	symbol tag (see below)

• Compact file format

The SMALL compiler generates output P-code as either a straightforward dump of the opcodes, or in a variable-length encoding similar to that of the MIDI "SMF" files. The "plain" encoding uses Little Endian for all opcodes as data words, meaning that a Big Endian processor should swap all cells that it reads from the P-code file before executing them. The alternative, "compact binary files", not only have a reduced size, the file format is also universal for Big Endian and Little Endian computers.

The header of the module (see page 89) and all tables (public functions, native functions, libraries public variables) are not compressed —these are always in Little Endian. The data that follows these tables is encoded with variable length codes: every four-byte cell is encoded in one to five bytes.

The highest bit of each byte is a "continuation" bit. If it is set, another bytes with seven more significant bits follows. The most significant 7 bits are stored first (at the lower file offset/memory address). When a series of bytes have been decoded, bit 6 (the next to most signification bit) of the first byte is repeated to fill the complete 32-bits.

Decoding examples:

`0x21`	`0x00000021`
`0x41`	`0xffffffc1`
`0x80 0x41`	`0x00000041`
`0x7f`	`0xffffffff`

• **Cross-platform support**

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.

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 Small 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 Small 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 SMALL 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 in de code segment (i.e., the address is relative to `COD`). 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 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.

- **Debugger support**

There is limited support for source level debuggers, built-in in the instruction set. These opcodes are not "regular" in the sense that they have more than one parameter.

The `size` parameter of the `FILE` and `SYMBOL` instructions gives the length of the instruction in *bytes*, excluding the bytes for the opcode and of the `size` field itself. The value of the `size` should always be a multiple of the size of a cell.

The `name` parameter of the `FILE` and `SYMBOL` instructions is a variable length, zero terminated string.

The `ord` parameter of the `FILE` and `LINE` instructions and the `off` parameter of the `SYMBOL` instruction are regular cell-sized parameters. The `line` parameter of the `LINE` instruction also has the size of a cell.

The `flg` parameter of the `SYMBOL` opcode holds the class and the type of the symbol. This parameter is a cell-sized value, but only the lowest two bytes are currently defined.

The type is in the lowest byte; it is one of the following values:

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 class is in the second byte; its value is:

0 the symbol refers to a global variable or to a function
1 the symbol refers to a local variable with a stack relative address
2 the symbol refers to a "static" local variable; the address is not stack relative

The "`off`" parameter is relative to either:

`COD` if the symbol refers to a function
`DAT` if the symbol refers to a global variable or a static local variable
`FRM` if the symbol refers to a local variable

An instruction for symbolic information is stored near the place where the variable or function to which it refers is created or declared. For local symbols, the symbolic information precedes the instructions that allocate, and optionally fill, the stack space for the variable(s). There is no run-time allocation for global symbols; therefore a symbolic debugger must browse through the code section to parse the symbolic information instructions and to collect the global symbols. This strategy was chosen as a compromise that minimized the overall effort to add symbolic debugging support to the compiler and to create a debugger. Writing the compiler was much easier when the symbolic information could be written where the variable was declared in the source code. A debugger should have a disassembler anyway. Combining these two resulted in decent debugger support with a low cost in terms of complexity.

The `SYMTAG` opcode extends a preceding `SYMBOL` instruction, by giving the "tag" value for the variable or function result. A debugger may use this value to look up the tag name in the "`tags`" table (in the "prefix", see figure 1 and page 89) and choose an appropriate display format. For example, a debugger may use

the `SYMTAG` code to automatically display floating point values as rational values, rather than (mis-)interpreting them as integers.

The `SRANGE` instruction extends a preceding `SYMBOL` instruction with information about the dimensions and the size of an array. The first parameter gives the dimension and the second parameter the length of that dimension.

- ◇ For single dimension arrays, a single `SRANGE` instruction follows the `SYMBOL` instruction. The first parameter of the `SRANGE` instruction is zero (0) and the second parameter gives the size of the array.
- ◇ For two-dimensional arrays, two `SRANGE` instructions complete the symbol definition. The first `SRANGE` instruction has its level (first parameter) set to one (1) and the second parameter set to the size of the major dimension. The second `SRANGE` instruction holds a level of zero and the size of the minor dimension.

When the "`size`" field of an `SRANGE` instruction is zero, the array size is indeterminate. When no `SRANGE` instruction follows a `SYMBOL` instruction that defines an array, the array should be assumed a single-dimensional array with an indeterminate size.

The Small compiler generates a `LINE` instruction before any other instruction for that line. The "`ord`" parameter is the file number to which the line relates. The Small compiler generates the `FILE` instruction at the point where the file is read. So a debugger would gather the filenames (and their ordinals) in the same way (and perhaps in the same phase) as the global symbols.

APPENDIX F

Code generation notes

The code generation of the SMALL compiler is fairly straightforward (also due to the simplicity of the abstract machine). A few points are worth mentioning:

◇ The abstract machine has instructions that the SMALL compiler currently does not generate. For example, the `LREF.pri` instruction works like the dereference operator ("*") in C/C++. SMALL does not support pointers directly, but references are just pointers in disguise. SMALL only supports references in function arguments, however, which means that the "pointer operations" in SMALL are always stack-relative. In other words, the SMALL compiler does *not* generate the `LREF.pri` instruction, although if *does* generate the `LREF.S.pri` instruction.

The abstract machine is fairly independent from the SMALL language, even though they were developed for each other. The SMALL language can easily grow in the future, possibly with a "reference" variable type, thereby giving the `LREF.pri` instruction a reason of being. The abstract machine cannot easily grow, however, because new instructions immediately make the new abstract machine incompatible with previous versions. That is, programs compiled for the new abstract machine won't run on the earlier release.

◇ For a native function, the SMALL compiler generates a `SYSREQ.C` instruction instead of the normal function call. The parameter of the `SYSREQ.C` instruction is an index in the native function table. A function in SMALL cleans up its arguments that were pushed on the stack, because it returns with the `RETN` instruction. The `SYSREQ.C` instruction does not remove items from the stack, so the SMALL compiler does this explicitly with a `STACK` instruction behind the `SYSREQ.C` instruction.

The arguments of a native function are pushed on the stack in the same manner as for a normal function.

In the "SMALL" implementation of the abstract machine (see page 6), the "system request" instructions are linked to the user-installed callback function. Thus, a native function in a SMALL program issues a call to a user-defined callback function in the abstract machine.

◇ At a function call, a SMALL 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 SMALL 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 SMALL 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 SMALL 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 SMALL 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 SMALL 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 SMALL are not drop through. At the end of every instruction in a `case` list, the SMALL compiler generates a jump to an "exit" label just after the `switch` instruction. The SMALL 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 SMALL 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 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 SMALL 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 SMALL 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.

APPENDIX G

Adding a garbage collector

SMALL uses only static allocation for all of its objects. The advantage of static allocation is that the memory requirements of a SMALL script are easy to determine (the SMALL 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 SMALL, a garbage collector is very convenient. This appendix describes how a garbage collector can be added to a host application that uses the SMALL 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 SMALL 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 SMALL 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 SMALL 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 SMALL 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 SMALL arrays is never necessary.

```
#define VERIFY(exp)     do if (!(exp)) abort(); while(0)

static cell AMX_NATIVE_CALL n_bstring(AMX *amx,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,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) {
    amx_GetAddr(amx, params[1], &cellptr);
    amx_SetString(cellptr, cstr, params[4], 0);
  } /* if */
  free(cstr);
  return 0;
}

static cell AMX_NATIVE_CALL n_bstrdup(AMX *amx,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,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 adaption of the simple program "`srun`" discussed at page 6.

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);        /* 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 SMALL 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.
- ◇ To call the garbage collector at a regular interval, monitoring the `DBG_LINE` opcode is the best option. However, this debug code will never be sent when the script was compiled without debug information.
- ◇ The debug hook does not consider system load, whereas you would want the garbage collection to take place especially when the system is not busy.
- ◇ The debug hook carries some overhead (though just a little).

That behind us, below is a debug hook that calls the garbage collector. It calls the garbage collector after executing every 100 lines and after each function return. Acting on the `DBG_RETURN` code circumvents problems for SMALL scripts that are compiled without debug information.

```
int AMXAPI srun_Monitor(AMX *amx)
{
  static int linecount;

  switch (amx->dbgcode) {
  case DBG_INIT:
    return AMX_ERR_NONE;

  case DBG_LINE:
    if (--linecount > 0)
      return AMX_ERR_NONE;
    linecount = 100;
    /* drop through */
  case DBG_RETURN:
    garbagecollect(amx, 1);
    return AMX_ERR_NONE;

  default:
    return AMX_ERR_DEBUG;
  } /* switch */
}
```

* The standard distribution comes with the source code for a minimal host application, in the subdirectory "amx/srun/" of where the toolkit was installed.

- **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 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.

APPENDIX H

License

The software toolkit "SMALL" (the compiler, the abstract machine and the documentation) are copyright ©1997–2005 by ITB CompuPhase. The Intel assembler implementation of the abstract machine and the just-in-time compiler (specifically the files AMXEXEC.ASM, JITR.ASM and JITS.ASM) are copyright ©1998-2003 Marc Peter. The file JITSN.ASM is translated from JITS.ASM and is partially copyright G.W.M. Vissers. The file AMXEXECN.ASM is translated from AMXEXEC.ASM and is partially copyright ITB CompuPhase.

SMALL is distributed under the "zLib/libpng" license, which is reproduced below:

This software is provided "as-is", without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1 The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2 Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3 This notice may not be removed or altered from any source distribution.

The zLib/libpng license has been approved by the "Open Source Initiative" organization.

Index

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