The SMite Forth Virtual Machine

Reuben Thomas

29th October 1993; revised 17th April 2018

Abstract

The design of the SMite Forth virtual machine is described. SMite's purpose is to provide an easily portable environment for ANS Forth compilers: to move a compiler from one system to another only SMite and the I/O libraries need be rewritten. Like most interpreters, SMite gains portability and compactness at the expense of speed, but it retains flexibility by providing instructions to call machine code and access the operating system.

Typographical notes

bForth instructions and SMite's registers are shown in `Typewriter` font; interface calls are shown in **Bold** type, and followed by empty parentheses. Quoted pronunciations of instructions and registers are given with components separated by dashes; single letters should be pronounced as the name of the letter.

Addresses are given in bytes and refer to SMite's address space except where stated. Addresses are written in hexadecimal; hex numbers are prefixed with "$".

1 Introduction

SMite is a simple virtual machine designed to enable the easy implementation of ANS Forth compilers, such as pForth, on different systems. It is a stack machine; the instruction set is based on the Core Word Set of ANS Forth [1]. This paper gives a full description of SMite, but certain implementation-dependent features, such as the size of the stacks, are purposely left unspecified, and the exact method of implementation is left to the implementor in many particulars.

SMite is self-contained. I/O can be implemented via extra instructions (see 3.13). Machine code routines on the host computer may be accessed using the `LINK` instruction. SMite supports a simple object module format.

SMite is conceptually (and usually in fact) a library, embedded in other programs. A small interface is provided for other programs to control SMite.

Since SMite is heavily oriented towards supporting Forth compilers, it is useful to understand how Forth compilers operate in order to understand SMite and to use it properly. An excellent introduction to Forth and Forth compilers is [3]. An overview of the language and its compilers is also provided in [1]. For an implementation of SMite, see [5].

2 Architecture

SMite's address unit is the byte, which is eight bits wide. Characters are one byte wide, and cells are four bytes wide. The cell is the size of the numbers and addresses on which SMite operates, and of the items placed on the stacks. The cell size is fixed to ensure compatibility of object code between implementations on different machines; the size of the address unit, character and cell has been chosen with a view to making efficient implementation of SMite possible on the vast majority of current machine architectures.

Cells may have the bytes stored in big-endian or little-endian order. The address of a cell is that of the byte in it with the lowest address.

2.1 Registers

The registers, each with its function and pronunciation, are set out in table 1.

Register	Pronunciation	Function
`EP`	"e-p"	The Execution Pointer. Points to the next cell from which an instruction word may be loaded.
`I`	"i"	The Instruction. Holds the opcode of an instruction to be executed.
`A`	"a"	The instruction Accumulator. Holds the opcodes of instructions to be executed, and immediate operands.
`MEMORY`	"memory"	The size in bytes of SMite's main memory, which must be a multiple of four.
`SP`	"s-p"	The data Stack Pointer.
`RP`	"r-p"	The Return stack Pointer.
`S0`	"s-nought"	The data Stack base.
`R0`	"r-nought"	The Return stack base.
`#S`	"hash-s"	The number of cells allocated for the data stack.
`#R`	"hash-r"	The number of cells allocated for the return stack.
`'THROW`	"tick-throw"	The address placed in `EP` by a `THROW` instruction.
`ENDISM`	"endism"	The endianness of SMite: 0 = Little-endian, 1 = Big-endian.
`CHECKED`	"checked"	0 = address checking off, 1 = address checking on.
`'BAD`	"tick-bad"	The contents of `EP` when the last exception was raised.
`-ADDRESS`	"not-address"	The last address which caused an address exception.

Table 1: SMite's registers

All of the registers are cell-wide quantities except for `I`, `ENDISM` and `CHECKED`, which are one byte wide.

To ease efficient implementation, the registers may only be accessed by bForth instructions (see section 3.8); not all registers are accessible, and only a few are writable.

2.2 Memory

SMite's memory is a discontiguous sequence of bytes with addresses in the range 0 to $2^{32} - 1$. Some locations may be read-only. The memory is contiguous in the range 0 to `MEMORY` $- 1$; this part is referred to as "main memory".

2.3 Stacks

The data and return stacks are cell-aligned LIFO stacks of cells. The stack pointers point to the top stack item on each stack. To **push** an item on to a stack means to store the item in the cell beyond the stack pointer and then adjust the pointer to point to it; to **pop** an item means to make the pointer point to the second item on the stack. Instructions that change the number of items on a stack implicitly pop their arguments and push their results.

The data stack is used for passing values to instructions and routines and the return stack for holding subroutine return addresses and the index and limit of the Forth `DO...LOOP` construct. The return stack may be used for other operations subject to the restrictions placed on it by its normal usage: it must be returned before an `EXIT` instruction to the state it was in directly after the corresponding `CALL`.

In what follows, for "the stack" read "the data stack"; the return stack is always mentioned explicitly.

2.4 Operation

Before SMite is started, `ENDISM` should be set to 0 or 1 according to the implementation. `CHECKED` should be set to 0 or 1 as desired. The other registers should be initialised as shown in table 2, except for `I`, which need not be initialised.

Register	Initial value
`EP`	$0
`A`	$0
`THROW`	$0
`'BAD`	$FFFFFFFF
`-ADDRESS`	$FFFFFFFF

Table 2: Registers with prescribed initial values

`MEMORY`, `ENDISM` and `CHECKED` must not change while SMite is executing.

SMite is started by a call to the interface calls **run()** or **single_step()** (see section 4.2). In the former case, the execution cycle is entered:

```
begin
    copy the least-significant byte of A to I
    shift A arithmetically 8 bits to the right
    execute the instruction in I
repeat
```

In the latter case, the contents of the execution loop is executed once, and control returns to the calling program.

The execution loop need not be implemented as a single loop; it is designed to be short enough that the contents of the loop can be appended to the code implementing each instruction.

Note that the calls **run()** and **single_step()** do not perform the initialisation specified above; that must be performed before calling them.

2.5 Termination

When SMite encounters a `HALT` instruction (see section 3.11), it returns the top data stack item as the reason code, unless `SP` does not point to a valid cell, in which case reason code -257 is returned (see section 2.6).

Reason codes which are also valid exception codes (either reserved (see section 2.6) or user exception codes) should not normally be used. This allows exception codes to be passed back by an exception handler to the calling program, so that the calling program can handle certain exceptions without confusing exception codes and reason codes.

2.6 Exceptions

When a `THROW` instruction (see section 3.11) is executed, an **exception** is said to have been **raised**. The exception code is the number on top of the stack at the time the exception is raised. Some exceptions are raised by other instructions, for example by `/` when division by zero is attempted; these push the exception code on to the stack and then execute a `THROW`.

Exception codes are signed numbers. -1 to -255 are reserved for ANS Forth exception codes, and -256 to -511 for SMite's own exception codes; the meanings of those that may be raised by SMite are shown in table 3. ANS Forth compilers may raise other exceptions in the range -1 to -255 and additionally reserve exceptions -512 to -4095 for their own exceptions (see [1, section 9.3.1]).

Code	Meaning
-9	Invalid address (see below).
-10	Division by zero attempted (see section 3.5).
-20	Attempt to write to a read-only memory location.
-23	Address alignment exception (see below).
-256	Illegal opcode (see section 3.14).

Table 3: Exceptions raised by SMite

Exception -9 is raised whenever an attempt is made to access an invalid address, either by an instruction, or during an instruction fetch (because `EP` contains an invalid address). Exception -23 is raised when a bForth instruction expecting an address of type *`a-addr`* (cell-aligned) is given a non-aligned address. When SMite raises an address exception (-9 or -23), the offending address is placed in `-ADDRESS`.

The initial values of `'BAD` and `-ADDRESS` are unlikely to be generated by an exception, so it may be assumed that if the initial values still hold no exception has yet occurred.

Address and alignment exceptions are only raised if `CHECKED` is 1. When `CHECKED` is 0, a faster implementation of SMite may be used.

If `SP` is unaligned when an exception is raised, or putting the code on the stack would cause `SP` to be out of range, the effect of a `HALT` with code -257 is performed (although the actual mechanics are not, as that too would involve putting a number on the stack). Similarly, if `'THROW` contains an invalid address, the effect of `HALT` with code -258 is performed.

3 Instruction set

The bForth instruction set is listed in sections 3.3 to 3.12, with the instructions grouped according to function. The instructions are given in the following format:

```
NAME            "pronunciation"                    ( before -- after )
                                                R: ( before -- after )
```

Description.

The first line consists of the name of the instruction followed by the pronunciation in quotes. On the right are the stack comment or comments. Underneath is the description. The two stack comments show the effect of the instruction on the data and return (R) stacks.

Stack comments are written

`( before -- after )`

where *before* and *after* are stack pictures showing the items on top of a stack before and after the instruction is executed (the change is called the **stack effect**). An instruction only affects the items shown in its stack comments. The brackets and dashes serve merely to delimit the stack comment and to separate *before* from *after*. **Stack pictures** are a representation of the top-most items on the stack, and are written

$$i_1 \ i_2 \ldots i_{n-1} \ i_n$$

where the i_k are stack items, each of which occupies a whole number of cells, with i_n being on top of the stack. The symbols denoting different types of stack item are shown in table 4.

Symbol	Data type
flag	flag
true	true flag
false	false flag
char	character
n	signed number
u	unsigned number
n|*u*	number (signed or unsigned)
x	unspecified cell
a-addr	cell-aligned address
c-addr	character-aligned address

Table 4: Types used in stack comments

Types are only used to indicate how instructions treat their arguments and results; SMite does not distinguish between stack items of different types. In stack pictures the most general argument types with which each instruction can be supplied are given; subtypes may be substituted. Using the phrase "$i \Rightarrow j$" to denote "i is a subtype of j", table 5 shows the subtype relationships. The subtype relation is transitive.

Numbers are represented in twos complement form. *c-addr* consists of all valid addresses. Numeric constants can be included in stack pictures, and are of type *n*|*u*.

Each type may be suffixed by a number in stack pictures; if the same combination of type and suffix appears more than once in a stack comment, it refers to identical stack items. Alternative *after* pictures are separated by "|", and the circumstances under which each occurs are detailed in the instruction description.

u ⇒ *x*
n ⇒ *x*
char ⇒ *u*
a-addr ⇒ *c-addr* ⇒ *u*
flag ⇒ *x*

Table 5: The subtype relation

The symbols *i*x*, *j*x* and *k*x* are used to denote different collections of zero or more cells of any data type. Ellipsis is used for indeterminate numbers of specified types of cell.

If an instruction does not modify the return stack, the corresponding stack picture is omitted. Some instructions have two forms, the latter ending in "I". This denotes Immediate addressing: the instruction's argument is included in the instruction cell (see section 3.1), rather than being placed separately in the next available cell.

3.1 Programming conventions

Since branch destinations must be cell-aligned, some instruction sequences may contain gaps. These must be padded with NEXT (opcode $00).

Literals and branch addresses should be placed in memory as follows. If a literal (see section 3.10) or branch address (see section 3.9) will fit in the rest of the cell directly after its instruction (see below), it should be placed there, and the immediate form of the instruction used. Otherwise it should be placed in the cell after the instruction. Further instructions may still be stored in the current cell. If more than one literal or branch instruction is encoded in one instruction cell, the literal values follow each other in successive cells.

Given an instruction cell with *n* bytes free, a literal will fit into it if it can be represented as an *n*-byte twos complement number. Immediate mode branch destinations are given as the relative cell count from the value EP will have when the instruction is executed (rather than the address of the instruction cell containing the instruction) to the address of the destination instruction cell (not as absolute addresses). The literal or branch is stored with the bytes in the same order as for a four-byte number, at the most significant end of the instruction cell.

3.2 Execution cycle

NEXT performs an instruction fetch when SMite runs out of instructions in the A register.

NEXT (--)

Load the cell pointed to by EP into A then add four to EP.

3.3 Stack manipulation

These instructions manage the data stack and move values between stacks.

DUP "dupe" (*x* -- *x x*)

Duplicate *x*.

DROP (*x* --)

Remove *x* from the stack.

`SWAP` (x_1 x_2 -- x_2 x_1)

Exchange the top two stack items.

`OVER` (x_1 x_2 -- x_1 x_2 x_1)

Place a copy of x_1 on top of the stack.

`ROT` "rote" (x_1 x_2 x_3 -- x_2 x_3 x_1)

Rotate the top three stack entries.

`-ROT` "not-rote" (x_1 x_2 x_3 -- x_3 x_1 x_2)

Perform the action of `ROT` twice.

`TUCK` (x_1 x_2 -- x_2 x_1 x_2)

Perform the action of `SWAP` followed by `OVER`.

`NIP` (x_1 x_2 -- x_2)

Perform the action of `SWAP` followed by `DROP`.

`PICK` ($x_u \ldots x_1$ x_0 u -- $x_u \ldots x_1$ x_0 x_u)

Remove u. Copy x_u to the top of the stack. If $u = 0$, `PICK` is equivalent to `DUP`. If there are fewer than $u + 2$ items on the stack before `PICK` is executed, the memory cell which would have been x_u were there $u + 2$ items is copied to the top of the stack.

`ROLL` (x_u $x_{u-1} \ldots x_0$ u -- $x_{u-1} \ldots x_0$ x_u)

Remove u. Rotate $u + 1$ items on the top of the stack. If $u = 0$ `ROLL` does nothing, and if $u = 1$ `ROLL` is equivalent to `SWAP`. If there are fewer than $u + 2$ items on the stack before `ROLL` is executed, the memory cells which would have been on the stack were there $u + 2$ items are rotated.

`>R` "to-r" (x --)
R: (-- x)

Move x to the return stack.

`R>` "r-from" (-- x)
R: (x --)

Move x from the return stack to the data stack.

`R@` "r-fetch" (-- x)
R: (x -- x)

Copy x from the return stack to the data stack.

3.4 Comparison

These words compare two numbers (or, for equality tests, any two cells) on the stack, returning a flag, true with all bits set if the test succeeds and false otherwise.

`<` "less-than" (n_1 n_2 -- *flag*)

flag is true if and only if n_1 is less than n_2.

`>` "greater-than" (n_1 n_2 -- *flag*)

flag is true if and only if n_1 is greater than n_2.

`=` "equals" (x_1 x_2 -- *flag*)

flag is true if and only if x_1 is bit-for-bit the same as x_2.

`U<` "u-less-than" (u_1 u_2 -- *flag*)

flag is true if and only if u_1 is less than u_2.

`U>` "u-greater-than" (u_1 u_2 -- *flag*)

flag is true if and only if u_1 is greater than u_2.

3.5 Arithmetic

These instructions consist of monadic and dyadic operators. All calculations are made without bounds or overflow checking, except as detailed for certain instructions.

Addition:

`+` "plus" ($n_1|u_1$ $n_2|u_2$ -- $n_3|u_3$)
Add $n_2|u_2$ to $n_1|u_1$, giving the sum $n_3|u_3$.

Multiplication and division (note that all division instructions raise exception -10 if division by zero is attempted, and round the quotient towards minus infinity, except for `S/REM`, which rounds the quotient towards zero):

`*` "star" ($n_1|u_1$ $n_2|u_2$ -- $n_3|u_3$)
Multiply $n_1|u_1$ by $n_2|u_2$ giving the product $n_3|u_3$.

`/` "slash" (n_1 n_2 -- n_3)
Divide n_1 by n_2, giving the single-cell quotient n_3.

`MOD` (n_1 n_2 -- n_3)
Divide n_1 by n_2, giving the single-cell remainder n_3.

`/MOD` "slash-mod" (n_1 n_2 -- n_3 n_4)
Divide n_1 by n_2, giving the single-cell remainder n_3 and the single-cell quotient n_4.

`U/MOD` "u-slash-mod" (u_1 u_2 -- u_3 u_4)
Divide u_1 by u_2, giving the single-cell remainder u_3 and the single-cell quotient u_4.

`S/REM` "s-slash-rem" (n_1 n_2 -- n_3 n_4)
Divide n_1 by n_2 using symmetric division, giving the single-cell remainder n_3 and the single-cell quotient n_4.

Sign function:

`NEGATE` (n_1 -- n_2)
Negate n_1, giving its arithmetic inverse n_2.

3.6 Logic and shifts

These instructions consist of bitwise logical operators and bitwise shifts. The result of performing the specified operation on the argument or arguments is left on the stack.

Logic functions:

`INVERT` (x_1 -- x_2)
Invert all bits of x_1, giving its logical inverse x_2.

`AND` (x_1 x_2 -- x_3)
x_3 is the bit-by-bit logical "and" of x_1 with x_2.

`OR` (x_1 x_2 -- x_3)
x_3 is the bit-by-bit inclusive-or of x_1 with x_2.

`XOR` "x-or" (x_1 x_2 -- x_3)

x_3 is the bit-by-bit exclusive-or of x_1 with x_2.

Shifts:

`LSHIFT` "l-shift" (x_1 u -- x_2)

Perform a logical left shift of u bit-places on x_1, giving x_2. Put zero into the least significant bits vacated by the shift. If u is greater than or equal to 32, x_2 is zero.

`RSHIFT` "r-shift" (x_1 u -- x_2)

Perform a logical right shift of u bit-places on x_1, giving x_2. Put zero into the most significant bits vacated by the shift. If u is greater than or equal to 32, x_2 is zero.

3.7 Memory

These instructions fetch and store cells and bytes to and from memory; there is also an instruction to add a number to another stored in memory.

`@` "fetch" (*a-addr* -- *x*)

x is the value stored at *a-addr*.

`!` "store" (*x a-addr* --)

Store *x* at *a-addr*.

`C@` "c-fetch" (*c-addr* -- *char*)

If `ENDISM` is 1, exclusive-or *c-addr* with 3. Fetch the character stored at *c-addr*. The unused high-order bits are all zeroes.

`C!` "c-store" (*char c-addr* --)

If `ENDISM` is 1, exclusive-or *c-addr* with 3. Store *char* at *c-addr*. Only one byte is transferred.

3.8 Registers

As mentioned in section 2.1, the stack pointers `SP` and `RP` may only be accessed through special instructions:

`SP@` "s-p-fetch" (-- *a-addr*)

a-addr is the value of `SP`.

`SP!` "s-p-store" (*a-addr* --)

Set `SP` to *a-addr*.

`RP@` "r-p-fetch" (-- *a-addr*)

a-addr is the value of `RP`.

`RP!` "r-p-store" (*a-addr* --)

Set `RP` to *a-addr*.

`EP@` "bracket-create" (-- *a-addr*)

Push `EP` on to the stack.

`S0@` "s-nought-fetch" (-- *a-addr*)

Push `S0` on to the stack.

`#S@` "hash-s-fetch" (-- *u*)

Push `#S` on to the stack.

R0@ "r-nought-fetch" (-- *a-addr*)

Push R0 on to the stack.

#R@ "hash-r-fetch" (-- *u*)

Push #R on to the stack.

'THROW@ "tick-throw-fetch" (-- *a-addr*)

Push 'THROW on to the stack.

'THROW! "tick-throw-store" (*a-addr* --)

Set 'THROW to *a-addr*.

MEMORY@ "memory-fetch" (-- *a-addr*)

Push MEMORY on to the stack.

'BAD@ "tick-bad-fetch" (-- *a-addr*)

Push 'BAD on to the stack.

-ADDRESS@ "not-address-fetch" (-- *a-addr*)

Push -ADDRESS on to the stack.

3.9 Control structures

These instructions implement unconditional and conditional branches, subroutine call and return, and various aspects of the Forth DO. . . LOOP construct.

Branches:

BRANCH (--)

Load EP from the cell it points to, then perform the action of NEXT.

BRANCHI "branch-i" (--)

Add A $\times$ 4 to EP, then perform the action of NEXT.

?BRANCH "question-branch" (*flag* --)

If *flag* is false then load EP from the cell it points to and perform the action of NEXT; otherwise add four to EP.

?BRANCHI "question-branch-i" (*flag* --)

If *flag* is false then add A $\times$ 4 to EP. Perform the action of NEXT.

EXECUTE ($a\text{-}addr_1$ --)
R: (-- $a\text{-}addr_2$)

Push EP on to the return stack, put $a\text{-}addr_1$ into EP, then perform the action of NEXT.

Subroutine call and return:

CALL (--)
R: (-- *a-addr*)

Push EP + 4 on to the return stack, then load EP from the cell it points to. Perform the action of NEXT.

CALLI "call-i" (--)
R: (-- *a-addr*)

Push EP on to the return stack, then add A $\times$ 4 to EP. Perform the action of NEXT.

EXIT (--)
R: (*a-addr* --)

Put *a-addr* into EP, then perform the action of NEXT.

3.10 Literals

These instructions encode literal values which are placed on the stack.

`(LITERAL)` "bracket-literal" `( -- x )`

Push the cell pointed to by `EP` on to the stack, then add four to `EP`.

`(LITERAL)I` "bracket-literal-i" `( -- x )`

Push the contents of `A` on to the stack. Perform the action of `NEXT`.

3.11 Exceptions

These instructions give access to SMite's exception mechanisms.

`THROW` "throw" `( -- )`

Put the contents of `EP` into `'BAD`, then load `EP` from `'THROW`. Perform the action of `NEXT`. If `'THROW` contains an out of range or unaligned address stop SMite, returning reason code -258 to the calling program (see section 4.2).

`HALT` `( x -- )`

Stop SMite, returning reason code *x* to the calling program (see section 4.2). If `SP` is out of range or unaligned, -257 is returned as the reason code.

3.12 External access

These instructions allow access to SMite's libraries, the operating system and native machine code.

`LINK` `( i*x -- )`

Make a subroutine call to the routine at the address given (in the host machine's format, padded out to a number of cells) on the data stack. The size and format of this address are machine-dependent.

3.13 Extra instructions

Instructions with opcodes from $80 to $FE inclusive may be implemented to provide system-dependent functionality, such as I/O.

3.14 Opcodes

In table 6 are listed the opcodes in numerical order. All undefined opcodes ($61–$FE) raise exception -256.

4 External interface

SMite's external interface comes in three parts. The calling interface allows SMite to be controlled by other programs. The `LINK` instruction and extra instructions mechanism allow implementations to provide access to system facilities, previously written code, code written in other languages, and the speed of machine code in time-critical situations. The object module format allows compiled code to be saved, reloaded and shared between systems. pForth is loaded from an object module.

Opcode	Instruction	Opcode	Instruction	Opcode	Instruction
$00	NEXT	$21	undefined	$42	EP@
$01	DUP	$22	undefined	$43	S0@
$02	DROP	$23	undefined	$44	#S
$03	SWAP	$24	undefined	$45	R0@
$04	OVER	$25	*	$46	#R
$05	ROT	$26	/	$47	'THROW@
$06	-ROT	$27	MOD	$48	'THROW!
$07	TUCK	$28	/MOD	$49	MEMORY@
$08	NIP	$29	U/MOD	$4A	'BAD@
$09	PICK	$2A	S/REM	$4B	-ADDRESS@
$0A	ROLL	$2B	undefined	$4C	BRANCH
$0B	undefined	$2C	undefined	$4D	BRANCHI
$0C	>R	$2D	undefined	$4E	?BRANCH
$0D	R>	$2E	NEGATE	$4F	?BRANCHI
$0E	R@	$2F	undefined	$50	EXECUTE
$0F	<	$30	undefined	$51	@EXECUTE
$10	>	$31	INVERT	$52	CALL
$11	=	$32	AND	$53	CALLI
$12	undefined	$33	OR	$54	EXIT
$13	undefined	$34	XOR	$55	undefined
$14	undefined	$35	LSHIFT	$56	undefined
$15	undefined	$36	RSHIFT	$57	undefined
$16	undefined	$37	undefined	$58	undefined
$17	U<	$38	undefined	$59	undefined
$18	U>	$39	@	$5A	undefined
$19	undefined	$3A	!	$5B	J
$1A	undefined	$3B	C@	$5C	(LITERAL)
$1B	undefined	$3C	C!	$5D	(LITERAL)I
$1C	undefined	$3D	undefined	$5E	THROW
$1D	undefined	$3E	SP@	$5F	HALT
$1E	+	$3F	SP!	$60	LINK
$1F	undefined	$40	RP@	$80–$FE	extra instructions
$20	undefined	$41	RP!	$FF	NEXT

Table 6: SMite's opcodes

4.1 Object module format

The first seven bytes of an object module should be the ASCII codes of the letters "smite" padded with ASCII NULs ($00), then the one-byte contents of the ENDISM register of the system which saved the module. The next four bytes should contain the number of cells the code occupies. The number must have the same endianness as that indicated in the previous byte. Then follows the code, which must fill a whole number of cells.

Object modules have a simple structure, as they are only intended for loading an initial memory image into SMite, such as the pForth compiler. Forth does not typically support the loading of compiled code into the compiler, nor there is any need, as compilers are fast, and an incremental style of program development, with only a little source code being recompiled at a time, is typically used.

4.2 Calling interface

The calling interface is difficult to specify with the same precision as the rest of SMite, as it may be implemented in any language. However, since only basic types are used, and the semantics are simple, it is expected that implementations in different language producing the same result will be easy to program. A Modula-like syntax is used to give the definitions here. Implementation-defined error codes must be documented, but are optional. All addresses passed as parameters must be cell-aligned. A SMite must provide the following calls:

native_address (*integer, boolean*) : pointer
: Return a native pointer corresponding to the given SMite address. If the SMite address is invalid, or the Boolean flag is true and the address is read-only, then a distinguished invalid pointer is returned.

run () : integer
: Start SMite by entering the execution cycle as described in section 2.4. If SMite ever executes a `HALT` instruction (see section 3.11), the reason code is returned as the result.

single_step () : integer
: Execute a single pass of the execution cycle, and return reason code -259, unless a `HALT` instruction was obeyed (see section 3.11), in which case the reason code passed to it is returned.

load_object (*file, address*) : integer
: Load the object module specified by *file*, which may be a filename or some other specifier, to the SMite address *address*. First the module's header is checked; if the first seven bytes are not as specified above in section 4.1, or the endianness value is not 0 or 1, then return -2. If the code will not fit into memory at the address given, or the address is out of range or unaligned, return -1. Otherwise load the bForth code into memory, converting it if the endianness value is different from the current value of `ENDISM`. The result is 0 if successful, and some other implementation-defined value if there is a filing system or other error.

SMite must also provide access to its registers and address space through appropriate data objects.

Acknowledgements

Leo Brodie's marvellous books [3, 2] turned my abstract enthusiasm for a mysterious language into actual knowledge and appreciation.

I have taken or extrapolated the pronunciations of Forth words from [1].

Martin Richards's demonstration of his BCPL-oriented Cintcode virtual machine [4] convinced me it was going to be fun working on virtual machines. He also supervised my BA dissertation project, Beetle, on which smite is based.

References

[1] American National Standards Institute. *ANS X3.215-1994: Programming Languages—Forth*, 1994.

[2] Leo Brodie. *Thinking FORTH*. Prentice-Hall, 1984.

[3] Leo Brodie. *Starting FORTH*. Prentice-Hall, second edition, 1987.

[4] Martin Richards. Cintcode distribution, 2000. `https://www.cl.cam.ac.uk/~mr/BCPL.html`.

[5] Reuben Thomas. An implementation of the SMite virtual machine for POSIX, 2018. `https://rrt.sc3d.org/`.