An implementation of the Beetle virtual machine for POSIX version 1.0rc3

Reuben Thomas

27th May 2018

1 Introduction

The Beetle virtual machine [3] provides a portable environment for the pForth Forth compiler [2], a compiler for ANSI Standard Forth [1]. To port pForth to a new CPU architecture or operating system, only Beetle need be rewritten. However, even this can be avoided if Beetle is itself written in ISO C, since almost all systems have an ISO C compiler available for them.

Writing Beetle in C necessarily leads to a loss of performance for a system which is already relatively slow by virtue of using a virtual machine rather than compiling native code. However, pForth is intended mainly as a didactic tool, offering a concrete Forth environment which may be used to explore the language, and particularly the implementation of the compiler, on a simple architecture designed to support Forth. Thus speed is not crucial, and on modern systems even a C implementation of Beetle can be expected to run at an acceptable speed.

As well as the virtual machine, C Beetle provides a debugger, which is described in [4].

The Beetle virtual machine is described in [3]. This paper only describes the features specific to this implementation.

2 Using C Beetle

This section describes how to compile C Beetle, and the exact manner in which the interface calls and Beetle's memory and registers should be accessed.

2.1 Configuration

Beetle is written in ISO C99 using POSIX-1.2001 APIs.

The Beetle virtual machine is inherently 32-bit, but will run happily on systems with larger (or smaller) addresses.

2.2 Compilation

Beetle's build system is written with GNU autotools, and the user needs only standard POSIX utilities to run it. Installation instructions are provided in the top-level file `README.md`.

2.3 Registers and memory

Beetle's registers are declared in `beetle.h`. Their names correspond to those given in [3, section 2.1], although some have been changed to meet the rules for C identifiers. C Beetle does not allocate any memory for Beetle, nor does it initialise any of the registers. C Beetle provides the interface call **init_beetle()** to do this (see section 2.5).

The variables EP, I, A, MEMORY, SP, RP, THROW, BAD and ADDRESS correspond exactly with the Beetle registers they represent, and may be read and assigned to accordingly, bearing in mind the restrictions on their use given in [3]. THROW, BAD and ADDRESS are mapped into Beetle's memory, so they are automatically updated when the corresponding memory locations are written to, and vice versa. CHECKED is the constant 1; it may be read but not assigned to.

C Beetle provides the ability to map native memory blocks into Beetle's address space; see below.

2.4 Extra instructions

C Beetle provides the following extra instruction.

2.4.1 Command-line arguments

Two calls are provided to access command-line arguments passed to C Beetle (excluding any that it interprets itself). They are copied from Gforth.

Opcode	Forth word	Stack effect	Description
$80	ARGC	`-- u`	the number of arguments
$81	ARG	`u1 -- c-addr u2`	the *u1*th argument

2.4.2 Standard I/O streams

These extra instructions provide access to POSIX standard input, output and error. Each call returns a corresponding file identifier.

Opcode	POSIX file descriptor
$82	STDIN_FILENO
$83	STDOUT_FILENO
$84	STDERR_FILENO

2.4.3 File system

The file system extra instructions correspond directly to ANS Forth words, as defined in [1].

Opcode	Forth word
$85	OPEN-FILE
$86	CLOSE-FILE
$87	READ-FILE
$88	WRITE-FILE
$89	FILE-POSITION
$8A	REPOSITION-FILE
$8B	FLUSH-FILE
$8C	RENAME-FILE
$8D	DELETE-FILE
$8E	FILE-SIZE
$8F	RESIZE-FILE
$90	FILE-STATUS

The implementation-dependent cell returned by FILE-STATUS contains the POSIX protection bits, given by the st_mode member of the struct stat returned for the given file descriptor.

File access methods are bit-masks, composed as follows:

Bit value	Meaning
1	read
2	write
4	binary mode

To create a file, set both read and write bits to zero when calling `OPEN-FILE`.

2.5 Using the interface calls

The operation of the specified interface calls is given in [3]. Here, the C prototypes corresponding to the idealised prototypes used in [3] are given.

Files to be loaded and saved are passed as C file descriptors. Thus, the calling program must itself open and close the files.

`uint8_t *native_address(UCELL address, bool writable)`
: Returns `NULL` when the address is invalid, or the writable flag is true and the address is read-only.

`CELL run(void)`
: The reason code returned by **run()** is a Beetle cell.

`CELL single_step(void)`
: The reason code returned by **single_step()** is a Beetle cell.

`int load_object(FILE *file, UCELL address)`
: If a file system error occurs, the return code is -3.

In addition to the required interface calls C Beetle provides an initialisation routine **init_beetle()** which, given a cell array and its size, initialises Beetle:

`int init_beetle(CELL *b_array, long size)`
: `size` is the size of `b_array` in *cells* (not bytes). The return value is -1 if `b_array` is `NULL`, and 0 otherwise. All the registers are initialised as per [3].

The following routines give easier access to Beetle's address space at the byte and cell level. On success, they return 0, and on failure, the relevant exception code.

`int beetle_load_cell(UCELL address, CELL *value)`
: Load the cell at the given address into the given `CELL *`.

`int beetle_store_cell(UCELL address, CELL value)`
: Store the given `CELL` value at the given address.

`int beetle_load_byte(UCELL address, BYTE *value)`
: Load the byte at the given address into the given `BYTE *`.

`int beetle_store_byte(UCELL address, BYTE value)`
: Store the given `BYTE` value at the given address.

The following routines give easier access to contiguous areas of Beetle's address space. On success, 0 is returned; if `to` if not less than `from`, or the addresses are not contained in the same area, or not writable if desired, -1 is returned; if either address is unaligned, or some other error occurs, a memory exception code is returned.

`int beetle_pre_dma(UCELL from, UCELL to, bool write)`
: Convert the given range to native byte order, so that it can be read (or written) directly.

`int beetle_post_dma(UCELL from, UCELL to)`
: Convert the given range to Beetle byte order, so that it can be used by Beetle after a direct access.

The following routines are provided to map system memory to Beetle's address space and vice versa:

`UCELL mem_here()`
: Returns the Beetle address at which the next mapping will be made.

`UCELL mem_allot(void *p, size_t n)`
: Map `n` bytes pointed to by `p` into Beetle's address space, and return the address mapped. Addresses are mapped sequentially.

`UCELL mem_align(void)`
: Like Forth's `ALIGN`, rounds up the address at which the next memory allocation will occur to the next cell.

The following routine allows the calling program to register command-line arguments that can be retrieved by the `ARG` and `ARGC` extra instructions.

`bool register_args(int argc, char *argv[])`
: Maps the given arguments register, which has the same format as that supplied to **main()**, into Beetle's memory. Returns `true` on success and `false` on failure (either because memory could not be allocated, or could not be mapped to Beetle's address space).

Programs which use C Beetle's interface must `#include` the header file `beetle.h` and be linked with the Beetle library. `beetle_opcodes.h`, which contains an enumeration type of Beetle's instruction set, and `beetle_debug.h`, which contains useful debugging functions such as disassembly, may also be useful; they are not documented here.

2.6 Other extras provided by C Beetle

C Beetle provides the following extra quantities and macro in `beetle.h` which are useful for programming with Beetle:

`B_TRUE`: a cell with all bits set, which Beetle uses as a true flag.

`B_FALSE`: a cell with all bits clear, which Beetle uses as a false flag.

`CELL_W`: the width of a cell in bytes (4).

`POINTER_W`: the width of a machine pointer in cells.

`NEXT`: a macro which performs the action of the `NEXT` instruction.

`CELL_pointer`: a union with members `CELL cells[POINTER_W]` and `void (*pointer)(void)`, which allow a function pointer suitable for the `LINK` instruction to be easily stored and retrieved. It is assumed that the pointer is pushed on to the stack starting with `cells[0]` and ending with `cells[POINTER_W − 1]`.

References

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

[2] Reuben Thomas. Beetle and pForth: a Forth virtual machine and compiler. BA dissertation, University of Cambridge, 1995. `https://rrt.sc3d.org/`.

[3] Reuben Thomas. The Beetle Forth virtual machine, 2018. https://rrt.sc3d.org/.

[4] Reuben Thomas. A simple debugger for the Beetle Forth virtual machine, 2018. https://rrt.sc3d.org/.