

An implementation of the Beetle virtual machine for POSIX version 3.0.1

Reuben Thomas

23rd September 2021

1 Introduction

The Beetle virtual machine [2] provides a portable environment for the pForth Forth compiler, 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 [3].

The Beetle virtual machine is described in [2]. 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 [2, 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 `beetle_init()` to do this (see section 2.5).

The variables `EP`, `I`, `A`, `M0`, `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 [2]. `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.

2.4 Extra library calls

C Beetle provides the following LIB calls.

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.

Number	Forth word	Stack effect	Description
0	ARGC	-- <i>u</i>	the number of arguments
1	ARG	<i>u1</i> -- <i>c-addr u2</i>	the <i>u1</i> th argument

2.4.2 Standard I/O streams

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

Number	POSIX file descriptor
2	STDIN_FILENO
3	STDOUT_FILENO
4	STDERR_FILENO

2.4.3 File system calls

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

Number	Forth word
5	OPEN-FILE
6	CLOSE-FILE
7	READ-FILE
8	WRITE-FILE
9	FILE-POSITION
10	REPOSITION-FILE
11	FLUSH-FILE
12	RENAME-FILE
13	DELETE-FILE
14	FILE-SIZE
15	RESIZE-FILE
16	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 [2]. Here, the C prototypes corresponding to the idealised prototypes used in [2] are given. The names are prefixed with **beetle_**.

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(beetle_UCELL address, bool writable)
    Returns NULL when the address is invalid, or the writable flag is true and the address is
    read-only.
```

```
beetle_CELL run(void)
    The reason code returned by beetle_run() is a Beetle cell.
```

```
beetle_CELL beetle_single_step(void)
    The reason code returned by beetle_single_step() is a Beetle cell.
```

```
int beetle_load_object(FILE *file, beetle_UCELL address)
    If a file system error occurs, the return code is -3. As an extension to the specification,
    if an object file starts with the bytes 35, 33 (!), then it is assumed to be the start of a
    UNIX-style “hash bang” line, and the file contents up to and including the first newline
    character (10) is ignored.
```

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

```
int beetle_init(beetle_CELL *b_array, size_t 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 [2].
```

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(beetle_UCELL address, beetle_CELL *value)
    Load the cell at the given address into the given beetle_CELL *.
```

```
int beetle_store_cell(beetle_UCELL address, beetle_CELL value)
    Store the given CELL value at the given address.
```

```
int beetle_load_byte(beetle_UCELL address, beetle_BYTE *value)
    Load the byte at the given address into the given beetle_BYTE *.
```

```
int beetle_store_byte(beetle_UCELL address, beetle_BYTE value)
    Store the given beetle_BYTE value at the given address.
```

The following routines allow direct native access to Beetle’s memory. On success, 0 is returned; if to if not less than *from*, or the addresses are not valid, -1 is returned; if either address is unaligned, or some other error occurs, a memory exception code is returned.

```
int beetle_pre_dma(beetle_UCELL from, beetle_UCELL to)
```

Convert the given range to native byte order, so that it can be read and written directly.

```
int beetle_post_dma(beetle_UCELL from, beetle_UCELL to)
```

Convert the given range to Beetle byte order, so that it can be used by Beetle after a direct access.

The following routine allows the calling program to register command-line arguments that can be retrieved by the ARG and ARGV LIB calls.

```
int beetle_register_args(int argc, const char *argv[])
```

Registers the given argument list, which has the same format as that supplied to **main()**. Returns 0 on success and -1 if memory could not be allocated.

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:

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

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

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

beetle_POINTER_W: the width of a machine pointer in cells.

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

beetle_CELL_pointer: a union with members `beetle_CELL cells[beetle_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[beetle_POINTER_W - 1]`.

References

- [1] American National Standards Institute. *ANS X3.215-1994: Programming Languages—Forth*, 1994.
- [2] Reuben Thomas. The Beetle Forth virtual machine, 2018. <https://rrt.sc3d.org/>.
- [3] Reuben Thomas. A simple shell for the Beetle Forth virtual machine, 2018. <https://rrt.sc3d.org/>.