An implementation of the Beetle virtual processor in ANSI C

Reuben Thomas

2nd April 1997

1 Introduction

The Beetle virtual processor [2] provides a portable environment for the pForth Forth compiler [2], a compiler for ANSI Standard Forth [1]. To move pForth between different machines and operating systems, only Beetle need be rewritten. However, even this can be avoided if Beetle is itself written in ANSI C, since almost all machines have an ANSI 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 processor 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.

C Beetle provides only the virtual processor, not a user interface. A simple user interface is described in [2].

The interface to an embedded Beetle is described in [2]. This paper only describes the features specific to this implementation.

2 Omissions

Certain features of Beetle cannot be rendered portably in C, and so have been left out of this implementation. Thus, this implementation does not fully meet the specification for an embedded Beetle.

The `OS` instruction is not implemented, as it depends on the operating system of the host machine, and this implementation of Beetle is meant to be portable. If executed, `OS` does nothing.

The interface call **save_standalone()** is not implemented, as it is difficult to implement portably without it merely using C Beetle to run an object file, which lacks the usual advantages of stand-alone programs, speed and compactness. For similar reasons, **load_library()** is not implemented either; the use of `LINK` to access C functions is recommended instead.

The recursion instructions `STEP` and `RUN` are not implemented, although they may be added in a future version.

Machine type	Symbol
MS DOS	`MSDOS`
RISC OS	`riscos`
Atari TOS	`atarist`
Unix	`unix`

Table 1: Supported machine types

3 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.

3.1 Configuration

It is impossible to write an ANSI C implementation of Beetle that will run unaltered on any machine. The few features that are machine-dependent are defined in a machine header file, which is included by `bportab.h`. The appropriate symbol for the machine on which C Beetle is to be compiled must be defined, and the machine header file will then be included automatically. The machine header files available at the time of writing are shown in table 1, together with the corresponding symbol that should be defined. These symbols are automatically defined by GNU C on the corresponding machines; if another compiler is used, the appropriate symbol should be defined. If C Beetle is to be compiled on a machine type not in the list, a new header file must be added to the directory `bportab`, modelled on the existing header files there, and `bportab.h` must be changed to load it.

The machine type `Unix` refers to most Unix machines. Systems tested successfully include System V Release 4, DEC OSF/1 and DEC ULTRIX. Some problems were encountered, which had effects ranging from impaired operation to non-compilation, but most are too machine and installation specific to be worth describing. The general observation may be made that when compiling on a 64-bit architecture many error messages may be generated by the compiler about pointer conversions. These occur because offsets into Beetle's address space are represented as four-byte numbers. As long as Beetle is never allocated more than 4Gb for its memory, this will not be a problem.

The following types must be defined in a machine header file:

`BYTE:` an unsigned eight-bit quantity (Beetle's byte).

`CELL:` a signed four-byte quantity (Beetle's cell).

`UCELL:` an unsigned four-byte quantity (an unsigned cell).

The following symbols should be defined if appropriate:

`BIG_ENDIAN` should be defined in the makefile to define the symbol of the same name whenever the C compiler is invoked if Beetle is compiled on a big-endian machine.

`FLOORED` should be defined if the C compiler performs floored division.

The following macros should also be defined:

`ARSHIFT(n, p)` should be set to a macro that assigns to `n` the result of shifting it right arithmetically `p` places, where `p` may range from 0 to 31.

`LINK` should be set to a macro that, calls the C function at the machine address held on top of Beetle's stack. This address will typically occupy one cell, but may occupy more. The macro must then alter `SP` so that the address is popped from the stack.

`GETCH` should be set to a macro that returns the next key-press without buffering and echo (like Curses's `getch()`).

`PUTCH(c)` should be set to a macro that prints the character `c` to `stdout` without buffering.

`NEWL` should be set to a macro that prints a `\n` on `stdout` without buffering.

The register `CHECKED` must be set (in `beetle.h`) at compile-time: set to one, address checking will be enabled, and set to zero it will be disabled. Its value cannot be altered at run-time. `MEMORY` can similarly be altered from its default value of 16384 if desired.

The C compiler must use twos-complement arithmetic. The settings in `bportab.h` are tested when the Beetle tests are run. If any is incorrect, the changes that should be made are listed.

3.2 Compilation

The utility Make is required to compile Beetle as supplied; this is available on most systems. First, edit the makefile, which is called `Makefile`, so that it will work with the C compiler and linker to be used. The variables `CCflags`, `Linkflags`, `CC` and `Link` may need to be changed. Then set `Touch` so that it will, when prepended to a filename, form a command that changes the timestamp of the file to the current time (on many systems, `touch` is the correct command).

Now run Make with `Makefile` as the makefile, and C Beetle should compile. To test the Beetle, run the script file `btests`. The Beetle object files can be made separately as the target `beetle`; the test programs can be made as the target `btests`.

3.3 Registers and memory

Beetle's registers are declared in `beetle.h`. Their names correspond to those given in [2, section A.2.1], although some have been changed to meet the requirements 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 3.4).

The variables `I`, `A`, `MEMORY`, `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] (e.g. copies of `BAD` and `ADDRESS` must be kept in Beetle's memory). `THROW` is a pointer to the Beetle register `'THROW`, so the expression `*THROW` may be used as the Beetle register. `CHECKED` is a constant expression which may be read but not assigned to.

EP, M0, SP and RP are machine pointers to the locations in Beetle's address space to which the corresponding Beetle registers point. Appropriate conversions (pointer addition or subtraction with M0) must therefore be made before using the value of one of these variables as a Beetle address, and when assigning a Beetle address to one of the corresponding registers. Examples of such conversions may be found in execute.c, where the bForth instructions are implemented.

The memory is accessed via M0, which points to the first byte of memory. Before Beetle is started by calling **run()** or **single_step()**, M0 must be set to point to a byte array which will be Beetle's memory.

3.4 Using the interface calls

The operation of the interface calls (except for **init_beetle()**) is given in [2]. Here, the C prototypes corresponding to the idealised prototypes used in [2] 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.

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

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

`int load_object(FILE *file, CELL *address)`
: If a filing error occurs, the return code is -3, which corresponds to a return value of EOF from **getc()**.

`int save_object(FILE *file, CELL *address, UCELL length)`
: If a filing error occurs, the return code is -3, which corresponds to a return value of EOF from **putc()**.

load_library() and **save_standalone()** are not implemented (see section 2).

In addition to the required interface calls C Beetle provides **init_beetle()** which, given a byte array, its size and an initial value for EP, initialises Beetle:

`int init_beetle(BYTE *b_array, long size, UCELL e0)`
: size is the length of b_array in *cells* (not bytes), and e0 is the Beetle address to which EP will be set. The return value is -1 if e0 is not aligned or out of range, and 0 otherwise. All the registers are initialised as per [2], and those held in Beetle's memory as well are copied there. Various tests are made to ensure that Beetle has compiled properly, and the program will stop and display diagnostic messages if not.

Programs which use C Beetle's interface must #include the header file beetle.h and be linked with the object files corresponding to the interface calls used; these are given in table 2. opcodes.h, which contains an enumeration type of Beetle's instruction set, and debug.h, which contains useful debugging functions such as disassembly, may also be useful; they are not documented here. (To use the functions in debug.h, link with debug.o.)

Interface call	Object file
run()	`run.o`
single_step()	`step.o`
load_object()	`loadobj.o`
save_object()	`saveobj.o`
init_beetle()	`storage.o` and `tests.o`

Table 2: Object files corresponding to interface calls

3.5 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).

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

References

[1] ANSI. *American National Standard 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. World Wide Web URL: `http://www.cl.cam.ac.uk/users/rrt1001/`.