Caml Virtual Machine — File & data formats
Document version: 1.4
http://cadmium.x9c.fr

February 6, 2010

Abstract: This document describes the binary formats used by Caml[1] in its "3.11.2" version for both bytecode files and marshalled data. This document is structured in two parts: the first one exposes the format of bytecode files, and the second one exposes the format of marshalled data.

Bytecode file format

The format of bytecode files is summarized by figure 1. Unused header is commonly os-executable code that looks for ocamlrun executable and launch it on the file. Trailer identifies the file as a caml bytecode file by magic ("Caml1999X008") and indicates the number of sections in the file. One should notice that datas and descriptors of sections do not need to be in the same order. All character and string values use the ISO-8859-1 encoding. The remainder of this section lists possible sections with their contents.

" `CODE` " section (mandatory) contains the bytecode to be executed. Its size must be a multiple of 4, as the code is composed of 4-byte integers (in little-endian representation). These integers are either insctructions bytecodes or instructions arguments. The list of intructions with related arguments is given in another document "Caml Virtual Machine – Instruction set" that can be downloaded at `http://cadmium.x9c.fr`.

" `DATA` " section (mandatory) contains the global data for the program, in the format defined in the second part of this document.

" `PRIM` " section (mandatory) contains a null-character-terminated list of null-character-terminated strings. Each string is the name of a primitive requested for program execution. The order of these strings defines the primitive integer identifiers: the first requested primitive is given the " `0` " integer identifier, the second one is given the " `1` " integer identifier, *etc.*

" `DLLS` " section contains a null-character-terminated list of null-character-terminated strings. Each string is the name of a linked library requested for program execution.

[1] The official Caml website can be reached at `caml.inria.fr` and contains the full development suite (compiler, tools, virual machine, *etc.*) as well as links to third-party contributions.

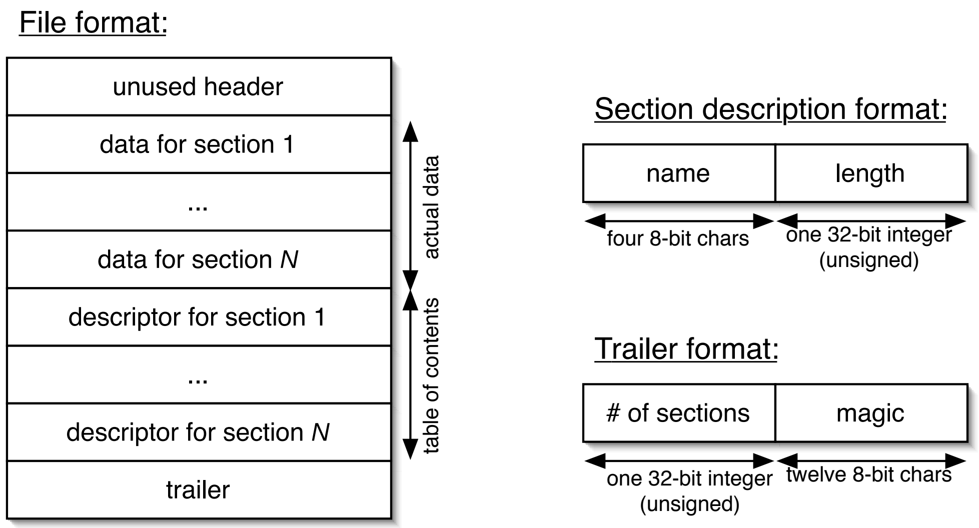

Figure 1: File format.

" DLPT " section contains a null-character-terminated list of null-character-terminated strings. Each string is a path for linked library search.

" DBUG " section contains an unsigned 32-bit integer indicating the number of debug elements. Elements follow, each being a couple containing an offset (as an unsigned 32-bit integer) and a marshalled value representing an `Instruct.debug-event` instance.

Marshalled data format

The format of marshalled values is summarized by figure 2. The following paragraphs give some precisions about particular data formats.

Integer values are stored in big-endian format. They encode values of the `int` type (`int32`, `int64` and `nativeint` types are coded as custom values).

String values are stored using ISO-8859-1 encoding.

Float values are stored using IEEE 754 encoding. According to code, values may be either big-endian (`0x0B`, `0x0D`, and `0x0F` codes) or little-endian (`0x0C`, `0x0E`, and `0x07` codes).

Code offset values (`0x10`) consist in the offset of a code address, relative to code start.

Block values are serialized as shown in figure 3. For atoms, no additional data needs to be stored as the tag is given by the header. Other blocks are stored by serializing their fields in ascending order, the size (number of blocks) being given by the header. Color is used by garbage collector and is set to zero in serialized data.

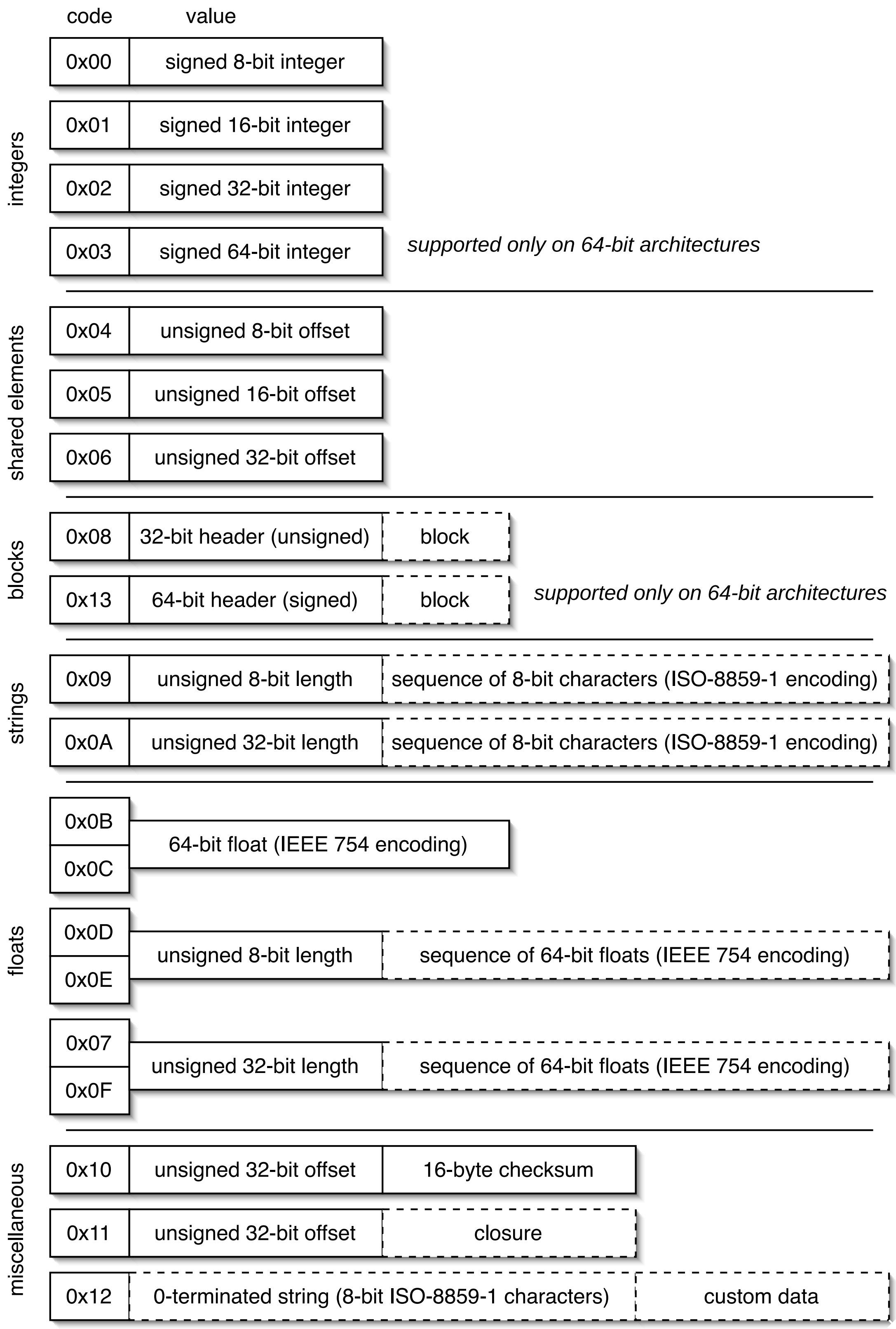

Figure 2: Data format.

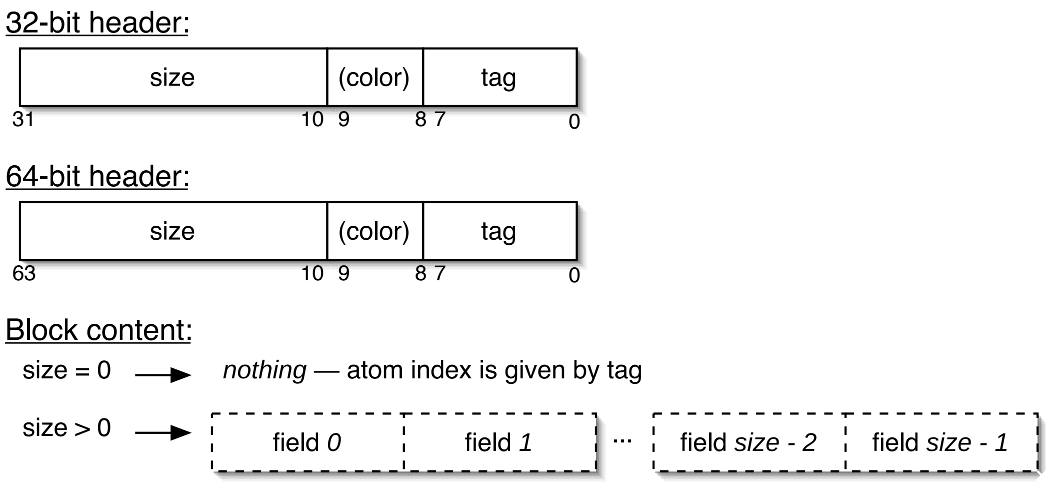

Figure 3: Block values.

Custom values are stored in two parts: the first one is the custom identifier (as a null-character-terminated string, using ISO-8859-1 encoding), the second one is custom-specific data.

Shared values are references to elements already (de)serialized of the current value. The offset defines this reference, zero pointing to the last read object, one pointing to the preceding object, *etc.*

Small elements are used to shorten value representation. They are stored using the specific encodig depicted in figure 4. A code from `0x20` to `0x3F` indicates a small string value, a code from `0x40` to `0x7F` indicates a small `int` value, and a code from `0x80` to `0xFF` indicates a small block value.

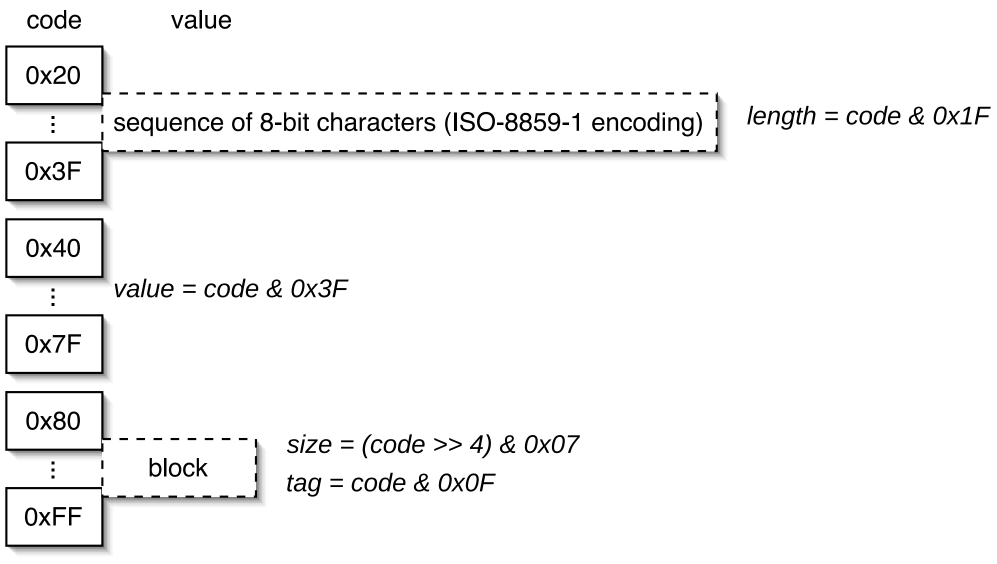

Figure 4: Small elements.