The Extremely Small File Format Specification (Draft)

January 2023

Abstract

This standard specifies the structure of *extremely small executable* and *extremely small loadable* files. The purpose of this document is to standardize the *extremely small* file format to promote correctness of such files.

Any additional details of the environment are not covered by this standard.

Table of Contents

Introduction

1 This is a draft version of the Extremely Small File Format specification, and there may be significant changes made to this standard before the final release. This draft version is only meant for those who are interested in contributing to the development of the Extremely Small File Format specification.

2 Footnotes are provided to clarify certain rules to the reader, as well as to mention the rationale behind the addition of such rules.

2 Footnotes and the annexes are informative.

1. Terms and definitions

1.1 byte
smallest addressable unit of storage

1.2 object
region of storage

1.2 alignment
requirement that an object is loaded at an address that is a multiple of a specified number of bytes

1.4 address
an integer that specifies the location of an object in the interpreter

1.5 interpretation
process of parsing data

1.6 data
sequence of bytes that denote one of: the magic number, the alignment of the code, the alignment of the heap, the size of the heap, the size of the stack, and the entry-point.

1.7 behavior
external appearance or action

1.8 stack
interpreter-defined entity or object

1.9 heap
interpreter-defined entity or object

1.10 execution
process of parsing *code* in an interpreter-defined way[1]

1.11 interpreter-defined
process that is not defined by the standard, rather is left for the interpreter to document

1.12 value
the result generated after a successful parse of data

1.13 load
place in memory

1 For example, the code may be passed on to the CPU for it to execute it.

2. Environments

1 For the purpose of describing how data shall be parsed, the standard describes two environments:

- the environment that writes the *extremely small executable* or *extremely small loadable* file (the *writer environment*), and
- the environment that parses data in that file (the *interpreter environment*).

2 Data written by the writer shall have the same value when interpreted by the interpreter[2].

3 Given a valid *extremely small file*, a standard-conforming interpreter is required to interpret it exactly as specified by this standard.

4 For an invalid *extremely small file*, an interpreter is required to quit the parsing process and produce at least one diagnostic message.

3. Limits

1 This standard imposes no restrictions on the number of bytes that can designate the code.

2 The value of the alignment of the code shall lie within the range 0 to 4294967295 inclusive.

3 The value of the alignment of the heap shall lie within the range 0 to 4294967295 inclusive.

4 The value of the size of the heap shall lie within the range 0 to 18446744073709551615 inclusive.

5 The value of the size of the stack shall lie within the range 0 to 18446744073709551615 inclusive.

6 The value of the entry-point shall lie within the range 0 to 18446744073709551615 inclusive.

4. Parsing

1 If parsing fails at any stage of the parsing process, the file being parsed is said to be invalid.

2 Before the parsing starts, the interpreter shall load the *extremely small executable* or *extremely small loadable* file at an 8 byte boundary.

4.1 Parsing a loaded extremely small executable file

1 An interpreter shall parse a loaded *extremely small executable* file in the following steps:

2 This is included to avoid endianness conflicts between the interpreter and the writer.

- The first 4 bytes are parsed. The result shall have the value 933631. This is called the *magic number*, which differentiates an *extremely small executable* file from other types of files. For any other value, the file is invalid and the parsing stops.
- The next 4 bytes designate the *alignment of the code*. It is the boundary the code will be aligned to when it is loaded by the interpreter. If the interpreter does not support such an alignment, the file being parsed is invalid and the parsing stops.
- The following 4 bytes designate the *alignment of the heap*. It is the boundary the heap will be aligned to when it is loaded by the interpreter. If the interpreter does not support such an alignment, the file being parsed is invalid and the parsing stops.
- Each of the next 4 padding bytes shall be unset. Otherwise, the behavior of the interpreter is not defined by this standard.
- The 8 bytes succeeding the padding bytes designate the *size of the heap*. It is the maximum size the heap can have. If the size of the heap exceeds this maximum size during execution, the behavior of the interpreter is not defined by this standard.
- The following 8 bytes designate the *size of the stack*. It is the maximum size the stack can have. If the size of the stack exceeds this maximum size during execution, the behavior of the interpreter is not defined by this standard.
- The next 8 bytes constitute the *entry-point*. It shall designate an address in the code where the control is transferred to when the execution starts.
- The next optional sequence of bytes constitutes the *code*.

4.2 Parsing a loaded extremely small loadable file

1 An interpreter shall parse a loaded *extremely small loadable* file in the following steps:

- The first 4 bytes are parsed. The result shall have the value 930303. This is called the *magic number*, which differentiates an *extremely small loadable* file from other types of files. For any other value, the file is invalid and the parsing stops.
- The next 4 bytes designate the *alignment of the code*. It is the boundary the code will be aligned to when it is loaded by the interpreter. If the interpreter does not support such an alignment, the file being parsed is invalid.
- The next 8 bytes constitute the *entry-point*. It shall designate an address in the code where the control is transferred to when the execution starts.
- The next optional sequence of bytes constitutes the *code*.

Annex A. Interpreter details (informative)

1 This annex tries to explain the relation between an interpreter, a writer and real-life implementations of those.

2 Any translator[3] that can generate a valid *extremely small executable* or *extremely small loadable* file can be considered to be a writer.

3 An operating environment[4] that can execute the translated file can be considered to be an interpreter.

3 Translator can be one of the entities specified in the following non-exhaustive list: a compiler, a transpiler.

4 An operating environment can be one of the entities specified in the following non-exhaustive list: an operating system, a virtual machine, a programming language interpreter.

Annex B. Extremely small file structure summary (informative)

1 This annex summarizes the *extremely small* file structure as described in 3.1 and 3.2.

2 The *extremely small executable* file structure:

- 4 bytes: magic number
- 4 bytes: alignment of the code
- 4 bytes: alignment of the heap
- 4 bytes: padding (unset)
- 8 bytes: size of the heap
- 8 bytes: size of the stack
- 8 bytes: entry point
- 0 or more bytes: code

3 The *extremely small loadable* file structure:

- 4 bytes: magic number
- 4 bytes: alignment of the code
- 8 bytes: entry point
- 0 or more bytes: code

Contributions

This specification would not have been possible without the contributions of the following people:

1 Somdipto Chakraborty