BEAM File Format

May 20, 2012

Contents

1 About this Document

This document describes BEAM (Bogdan's Erlang Abstract Machine) as included in the Erlang Open Telecom Platform (Erlang/OTP).[1] It is intended to provide useful information on the BEAM file format and the Virtual Machine.

More information on Erlang can be found at http://www.erlang.org/

1.1 Status

This is an INCOMPLETE DRAFT document, documenting the BEAM Virtual Machine included in version R15B01 of Erlang/OTP. This is draft 1 of this document.

This document makes statements about what must or must not occur in a BEAM file, and that is based on analysis of code both in erlc and the BEAM VM. References to code are provided where possible. None of these statements have been tested by altering BEAM byte code and checking the results through either the BEAM VM or the BEAM disassembly capabilities of the Erlang language. To provide additional proof of these assertions, they should be tested, and reliance on this document without those tests is taken at the reader's risk. You have been warned.

2 Introduction

2.1 Background

The BEAM File format was originally based off Electronic Arts Interchange File Format 85 (EA IFF 85).[2] [3]

2.2 Conventions

2.2.1 Key Words

This document adopts the language of Internet Engineering Task Force (IETF) Request for Comments (RFC) 2119, "Key words for use in RFCs to Indicate Requirement Levels".[4] This is done to ensure appropriate comprehension about absolute requirements, situations that exist under certain conditions, and items that are optional. There is one caveat to this adoption: While the definitions of the key words shown there are adopted, this document does not attempt to capitalize those key words (in the hope of providing better readability).

2.2.2 Bit Structure

All numbers stored in BEAM files are big-endian unless otherwise specified.

When a specific bit is referred to by the bit number, the 0-bit is the least significant bit. This will include when bits are shown as variables with an index as a subscript. The least-significant bit will appear on the right, unless otherwise specified. The example of encoding the number 10 is shown below.

X_3	X_2	X_1	X_0
1	0	1	0

[1]More information on the history of BEAM and the VMs used in Erlang can be found at http://www.erlang-factory.com/upload/presentations/247/erlang_vm_1.pdf

[2]http://www.martinreddy.net/gfx/2d/IFF.txt

[3]http://www.erlang.se/~bjorn/beam_file_format.html

[4]http://www.ietf.org/rfc/rfc2119.txt

2.2.3 Function Notation

This document adopts the "Erlang style" of function identification.[5] In other words, a function can be uniquely identified with 3 pieces of information: The module in which the function resides, the function name, and the arity of the function. (The arity is the number of arguments a function has). We write these as:
`module:function/arity`

A specific example for the cosine function (abbreviated as 'cos') provided in a module called math:
`math:cos/1`

2.2.4 Font Styles

A `Fixed Width` font is used to identify short code samples or function names.

2.3 Key Definitions

For purposes of this document, some level of basic programming knowledge is assumed. No attempt is made to define common words like function, module, or other concepts that are pervasive. However, a number of concepts are not as well known, but are critical to understanding BEAM files. These concepts are pervasive in BEAM files and VMs designed to interpret them, such as the BEAM VM provided as part of Erlang/OTP. Definitions for those items are presented here.

2.3.1 Atom

An Atom is a non-numerical constant. These are similar to the concept of enumerated types in programming languages like Java or C++.

In BEAM files, Atoms are global. A Virtual Machine that interprets BEAM files must treat Atoms as global. Therefore, if 'January' is defined as an Atom in two BEAM files, those atoms are equal to the Virtual Machine (and would pass what most developers would recognize as an equality test [==]).

Note: Restrictions on Atoms may exist in programming languages that compile to BEAM format, but since this is now a binary file and items will be specifically identified as atoms, there is less restriction on what characters may appear in an Atom, and there are no escape sequences in the storage of an Atom in a BEAM file.[6]

3 File Structure

The EA IFF 85 standard defines a header for the file, followed by chunks. Chunks are a byte-stream, each with a header and a variable-length data structure (the length of which is defined in the chunk's header). Each chunk must be a multiple of 4 bytes long. The chunk must be padded to a multiple of 4 bytes if the native data within the chunk is not a multiple of 4 bytes in length.[7] The first 4 bytes of the header of a chunk is a magical value identifying the type of chunk. This document calls that magical value the chunk Identifier.

The data within the chunks is further structured. Tables are a group of zero or more fixed- or variable-sized data structures (depending on the chunk type). Each of the items in a Table is referred to in this document as a Record.

Note: The use in this document of the 'Table' and 'Record' key words may make the names of the items in the BEAM file inconsistent with the names that appear in the Erlang Compiler (erlc) and the BEAM Virtual Machine. In some cases that is unavoidable (as those two already call the chunk by a different

[5] http://www.erlang.org/doc/reference_manual/users_guide.html version 5.9.1, Section 5.1
[6] TODO: Check restrictions on Atoms - what characters are not legal (if any)? Null prohibited? Others?
[7] TODO: Defensive test to check if the padding MUST be zeros or SHOULD be zeros?

name), while in others, a standardized name is adopted for this document in order to leverage explanatory key words.

For fixed-length records, the table is written as a contiguous data structure. Each record will not have an individual header.

For variable-length records, there are two side effects: First, each record will require a header to indicate the length of the record. Second, it is not possible to pre-determine the location of all the records in the table (The entire table must be read to determine the value of the last record).

Some chunks must appear in the BEAM file, others are optional. If a specific type of chunk appears, it must only appear one time. Although it is unlikely one would ever reach this size, the maximum BEAM file size is approximately 2GB (this is due to the form length appearing in the header as a 32-bit number of bytes).

The following chunks must appear in a BEAM file:

Identifier	Chunk Name	Description
Atom	Atom Table Chunk	Defines Atoms used in the file
Code	Code Chunk	Defines Code to execute functions in the file
StrT	String Table Chunk	Defines Strings used in the file[8]
ImpT	Import Table Chunk	Defines methods that are implrted into (called from) the module
ExpT	Export Table Chunk	Defines methods that are exported from the module

The following chunks must appear in a BEAM file if they are required by references made in the Code Chunk:

Identifier	Chunk Name	Description
FunT	Local Function Table Chunk[9]	useful for cross-reference tools
LitT	Literal Table Chunk	Stores Literals (like large integers, lists, etc.)
Line	Line Table Chunk	Defines information about line numbers from the original source file

The following chunks are optional chunks for a BEAM file.

Attr	Attribute Chunk	defines the attributes of the module
CInf	Compile Information Chunk	defines the compile information about the module
Trac	Function Trace Chunk[10]	If present, the loaders will insert special trace instructions to support function tracing
AbsT	Abstract Code Chunk	Contains an abstract representation of the code, useful for debugging, ignored by VM
LocT	TBD	TBD

Chunks in the BEAM file are not order-dependent, the loading code of a virtual machine must ensure any loading dependencies are resolved.[11]

3.1 (Presumed) Design Considerations

A number of the Tables within BEAM files serve as reference for the Code Chunk. This is to allow the Code Chuck to remain compact, and to avoid storing duplicate information when a value is reused. Each of the types of potential reused values is stored in a different table. For those with appropriate experience, these can loosely be thought of like tables in a relational database.

BEAM files contain some 'hint' information for a program that is loading a BEAM file. While these values not formally necessary to load a BEAM file (in a mathematical sense they are duplicate information), this 'hint' information is provided to help with memory allocation. These 'hint' fields must not be ignored

[8]TODO: how is this different than LitT?

[9]This is called the Lambda Chunk by the BEAM VM

[10]TODO: Is it possible for Trac length to be zero? (currently is loaded as 4 with 4 '00' bytes when no Trac)

[11]This appears true from inspection, but should be tested

when BEAM files are generated, and these fields must contain the exact value appropriate for the given field. This will ensure both that sufficient memory is allocated by a program loading the BEAM file and that a program loading a BEAM file can rely on these fields for looping and validation.

3.2 References

When an item defined within one of the tables is referred to by the code, this may be called a reference. The references are of variable length, designed to make the most common references very small in the BEAM file, while accommodating larger values when necessary.[12]

The minimum size of a reference is 1 byte. The 1-byte form of a reference is shown here:

IX_3	IX_2	IX_1	IX_0	0	ID_2	ID_1	ID_0

In this case, the 3-bit value of Identifier ID is $0 < ID < 6$. For ID == 7, the record is variable length (see below). The value IX is used as the index within the appropriate namespace of the identifier.[13]

The 2-byte form of a reference is shown here:

IX_{10}	IX_9	IX_8	0	1	ID_2	ID_1	ID_0
IX_7	IX_6	IX_5	IX_4	IX_3	IX_2	IX_1	IX_0

In this case, the 3-bit value of Identifier ID is $0 < ID < 6$. For ID == 7, the record is variable length (see below). The value IX is used as the index within the appropriate namespace of the identifier.

Note that the maximum value of any reference index in a BEAM file is 2047. This is not meant to imply that all reference types are valid up to 2047. Some may be more limited (for example, each process has 1024 X registers[14])

For ID == 7, the record header (the first byte) is:

L_2	L_1	L_0	x	x	1	1	1

Where L is the length (in bytes) of the rest of the record.[15]

3.2.1 Reference Namespace Identifiers

Identifier	Type	Assembler ID	Description
00	Literal[16]	u	Index to a record appearing in the Literal Chunk
01	Integer	i	An integer
02	Atom	a	Index to a record appearing in the Atom Chunk
03	X Register	x	Index to an X Register in the Virtual Machine
04	Y Register	y	Index to a Y Register in the Virtual Machine
05	Label (i.e. Function)	f	Index to a Label in Code Chunk
06	Character[17]	h	
07	Extended Format	z	(List, Float, etc.)

Atom 0 is the special atom NIL[18]

X and Y are zero-indexed (there IS an x,0 and a y,0 register)

[12]See beam_load.c #define GetTagAndValue

[13]TODO: I beleive it would be beneficial to have an example of using a reference/index in a lookup of a table. Perhaps this is part of a larger example at some point?

[14]http://dl.dropbox.com/u/4764922/beam.pdf

[15]TODO: Need to decode the rest of the 'extended' ID == 7 format.

[16]TODO: Need to check what index Literal Table uses (is 00 a special value? - doesn't look like it from inspection, but should make an explicit test)

[17]Cannot be generated by the current version of erlc. Not yet tested if BEAM VM still supports use of this Identifier

[18]TODO: Find reference for this information - it needs it and I didn't write it down in the first pass

Label is 1 indexed for the creator of a BEAM file, the 0 value is special: it contains the undefined location[19]

Character must be a value from 0<c<65535 (must be a valid ISO 8851-1 character).

3.3 Form Header

The Form Header is a 12-byte section at the beginning of the file. This is the structure of the Form Header:

Length	Value	Description
4 bytes	'FOR1'	Magic number indicating the IFF form. This is an extention to IFF indicating that all chunks are four-byte aligned.
4 bytes	length	Form length (file length in bytes - 8)
4 bytes	'BEAM'	Form type (Magic number for BEAM files)

The form header is followed by the chunks.

3.4 The Atom Chunk

The Atom Chunk stores Atoms - the constant vales (including identifiers like the module name and function names). The Atom Chunk is composed of a Header followed by Atom definitions.

3.4.1 Header

The Atom Chunk Header is composed of 12 bytes. This is the structure of the Atom Chunk Header:

Length	Value	Description
4 bytes	'Atom'	Magic number indicating the Atom Chunk.
4 bytes	size	Atom Chunk length in bytes
4 bytes	count	Number of atoms in the Atom Chunk

3.4.2 Atom Definition Format

Each Atom Definition is composed of two items: This is the structure of the Atom Definition:

Length	Value	Description
1 bytes	length	Number of bytes in the Atom
n bytes	atom	The Atom (in the form of ASCII characters).

Restriction: Atoms are limited to 255 characters or less. Characters must be valid characters in ISO 8859-1.

3.4.3 Requirements

The first Atom to appear in the Atom Chunk must be the module name.

3.5 The Code Chunk

The Code Chunk is a mandatory chunk that stores the code for the module. The Code Chunk is composed of a header followed by one or more function definitions.[20] [21]

[19]This is in beam_load.c

[20]TODO: Need to check whether the module_info items are required in BEAM files? By indications so far, they are not required as they are just delegations to functions in the erlang module

[21]TODO: Check if code must to "belong to" a function. While elc will not produce it, can a well structured BEAM file start with code that is not part of a function or will the VM loader error on it? If that code is not preceded by a label, then it will produce code that is unreachable. Is that illegal or just wasteful?

3.5.1 Header

The Code Chunk Header is composed of 28 or more bytes. This is the structure of the Code Chunk Header:

Length	Value	Description
4 bytes	‘Code’	Magic number indicating the Code Chunk.
4 bytes	size	Code Chunk length in bytes
4 bytes	info-size[22]	Length of the information fields before code. This is for future expansion. It must be set to zero when the instruction set identifier is 0.
4 bytes	version	Instruction set version. This is 0 for OTP R5–R15. This will be incremented if the instructions are changed in incompatible ways (instructions renumbered, argument types changed, etc.)
4 bytes	opcode-max	The highest opcode used in the code section. This allows addition of new opcodes without incrementing the version of the BEAM file.
4 bytes	labels[23]	The number of labels. This is a hint for the loader to help allocate the label table.
4 bytes	function-count	The number of functions contained in the Code Chunk.

Note: The value in the labels field holding the number of labels must be 1 greater than the number of labels in the BEAM file (remembering that labels are indexed from zero). This is because the 0 label is reserved for the undefined location. The compiler must set the labels value equal to the highest label it uses + 1.

3.5.2 Code Definition Format

The code is simply defined as a block of code. Special instructions identify where functions insert into the code.[24]

Each operation is coded according to the opcodes defined in the (not yet existing) Opcode section

3.5.3 Requirements

A BEAM file loader should not load a BEAM file if: (a) It does not understand the instruction set version of the BEAM file (b) The opcode-max is higher than the greatest opcode that the loader comprehends

3.6 String Table Chunk

A String Table Chunk is used when there are string values to be referenced in the BEAM file.[25]

[22]TODO: Need to understand purpose of information fields, and what a loader should do if it encounters information fields that it does not understand - are information fields supposed to be required or are they optional information, or ?? Need to establish this in order to make MUST vs SHOULD statements

[23]TODO: As defensive checking, should check whether labels can go unused, and if a loader would complain in any way. In that way we can define if MUST use all labels implied by the labels field or SHOULD not leave labels unused

[24]TODO: This is about behavior of the opcodes, but important to answer for code structure in BEAM: Defensive test to see if it's possible to have fallthrough in such a way that a function does not return but rather passes through a label or passes through a func_info. (e.g. are the function_info and label calls definitional only or does it actually insert some instructions into the stream?)

[25]TODO: Is this still used? Can it be written? Would like example code that produces a non-null String Table Chunk.

3.6.1 Header

The String Chunk Header is composed of 8 bytes. This is the structure of the String Chunk Header:

Length	Value	Description
4 bytes	'StrT'	Magic number indicating the String Table Chunk.
4 bytes	size	String Table Chunk length in bytes

3.6.2 String Definition Format

TODO: Format to be validated after a code example is found.

3.7 Import Table Chunk

The Import Table Chunk is used to define methods in other modules that are called from the current module.

3.7.1 Header

The Import Table Chunk Header is composed of 8 bytes. This is the structure of the Import Table Chunk Header:

Length	Value	Description
4 bytes	'ImpT'	Magic number indicating the Import Table Chunk.
4 bytes	count	Import Table Chunk length in the number of records

Note each record is 12 bytes. The size of the data for the Import Table Chunk will be equal to 12 bytes * number of records

3.7.2 Import Record Format

The Import Record Format is a fixed format of 12 bytes per record. This is the structure of the Import Record Format:

Length	Value	Description
4 bytes	module-atom-id	An identifier of the atom used to identify the module from which the method is imported. The identifier is used to look up the Atom in the Atom Chunk of this BEAM file.
4 bytes	method-atom-id	An identifier of the atom used to identify the method name of the method to be imported. The identifier is used to look up the Atom in the Atom Chunk of this BEAM file.
4 bytes	arity	The arity of the method to be imported

Together these 3 items contain the information we would normally associate with a method signature, the module, function name, and arity, such as: `math:cos/1` where
`math` is the module
`cos` is the function name
`1` is the arity.

3.8 Export Table Chunk

The Export Table Chunk is used to define methods that this module exports, and therefore those methods which can be called by other modules.

3.8.1 Header

The Export Table Chunk Header is composed of 8 bytes. This is the structure of the Export Table Chunk Header:

Length	Value	Description
4 bytes	'ExpT'	Magic number indicating the Export Table Chunk.
4 bytes	count	Export Table Chunk length in the number of records

Note each record is 12 bytes. The size of the data for the Export Table Chunk will be equal to 12 bytes * number of records

3.8.2 Export Record Format

The Export Record Format is a fixed format of 12 bytes per record. This is the structure of the Export Record Format:

Length	Value	Description
4 bytes	method-atom-id	An identifier of the atom used to identify the method name of the method to be exported. The identifier is used to look up the Atom in the Atom Chunk of this BEAM file. Note that for purposes of this index, the list of Atoms in the Atom Chunk is an index *starting at one*.
4 bytes	arity	The arity of the method to be exported
4 bytes	module-label	The code label identifing the starting position of the method within the Code Chunk of the BEAM file.

Together these 3 items contain the information required to being execution at the appropriate point. We need to sets of information to begin execution of a method. We need the method signature we are familiar with: `math:cos/1` where
`math` is the module
`cos` is the function name
`1` is the arity.
...and the location (label) in the code block to begin execution. The module name is not required in the export table, since the module name is known from the first entry in the Atom Chunk. It is shared among all of the exported methods. The other required information is contained within the Export Record.

3.9 Literal Table Chunk

The Literal Table Chunk is used to define literals used in the module. These literals are stored in a compressed state in the BEAM file.

3.9.1 Header

The Literal Table Chunk Header is composed of 8 bytes. This is the structure of the Literal Table Chunk Header:

Length	Value	Description
4 bytes	'LitT'	Magic number indicating the Literal Table Chunk.
4 bytes	bytes	Compressed Literal Table length in bytes

Note the bytes provided above is the bytes of the compressed record.

3.9.2 Compressed Literal Table Format

The Literal Table Chunk contains a single Compressed Literal Table. This Compressed Literal Table contains a 4-byte header followed by the compressed data. The header is a hint to the loader that indicates the uncompressed size of the compressed data in the Compressed Literal Table. This is the structure of the Compressed Literal Table Header:

Length	Value	Description
4 bytes	bytes	Uncompressed Literal Table Length in bytes

The compression is done with zlib. Once uncompressed, the result is the Uncompressed Literal Table.

3.9.3 Uncompressed Literal Table Format

The Uncompressed Literal Table contains one or more literal records. It contains a 4-byte header followed by the records. The header indicates the number of literal records contained in the Uncompressed Literal Table. This is the structure of the Uncompressed Literal Table Header:

Length	Value	Description
4 bytes	records	Number of Literal Records in the Uncompressed Literal Table

3.9.4 Literal Record Format

Each record in the Uncompressed Literal Table is composed of a header and data. This is the structure of the Literal Record Format:

Length	Value	Description
4 bytes	n	Number of bytes in the Literal
n bytes	data	Data used to store the Literal. This is in external format

3.10 Line Table Chunk

A Line Table Chunk is used to help trace line numbers within the original code file.

3.10.1 Header

The Line Table Chunk Header is composed of 8 bytes. This is the structure of the Line Table Chunk Header:

Length	Value	Description
4 bytes	'Line'	Magic number indicating the Line Table Chunk.
4 bytes	size	Line Table Chunk length in bytes

3.10.2 Line Table

The Line Table has a 20-byte header. The data stored in the header is: This is the structure of the Line Table Header:

Length	Value	Description
4 bytes	version	Line Table Version (curently must be set to zero)
4 bytes	flags	Reserved for future use = ignore
4 bytes	count	Number of Line instructions in the Code Chunk
4 bytes	records	Number of Line Records in the Line Table (the data that follows the Line Table Header)
4 bytes	fnames[26]	Number of File Name Records in the Line Table

The Line Table header is followed by the Line Records. Each record is coded in the Argument Encoding format.

The Line Records are followed by File Name Records. Each record is coded as a 16-bit integer length, followed by the number of bytes provided in the length (presumably of ISO 8859-1 characters).

Note: The Line items table is actually one-indexed for user use, the 0th item in that table is special and is used to store the undefined location.[27]

4 External Format

The external format, used in the Literal Table Chunk, is documented at
http://www.erlang.org/doc/apps/erts/erl_ext_dist.html

5 Key Acronyms

BIF: Built-in functions. These are functions built into the vitual machine. These are effectively part of Erlang, and are (mostly) considered part of the erlang module.

6 Potential Work Items

The definition of Tables in BEAM files are overloaded. This is because both fixed-length records (some might call these arrays) and variable length records (what might be best called tables) are called Tables. This is not necessarily a problem, and calling ExpT the Export Array might be more trouble than it's worth. So the key question is a very specific language syntax question: Do we want to define fixed-length things as arrays to try to help clarify fixed vs. variable length?

[26]TODO: Establish how File Name Records would be created... since they are not present in default compilation by erlc (at least for stuff I've written/analyzed)

[27]TODO: Check if this is BEAM VM specific or if this is intended to be a generic characteristic of BEAM files - not quite sure yet how to call this one.