L2/01-373

DRAFT INTERNATIONAL STANDARD ISO/IEC DIS 20970

Attributed to ISO/IEC JTC 1 by the Central Secretariat (see page ii)

Voting begins on
2001-05-31

Voting terminates on
2001-11-31

PUBLICLY AVAILABLE SPECIFICATION PROCEDURE

Information technology — Programming languages, their environments and system software interfaces — JEFF file format

Technologies de l'information — Langages de programmation, leurs environements et interfaces de logiciel système — Format de fichier JEFF

ICS 35.060

This Publicly Available Specification (PAS) is being submitted for Fast-track processing in accordance with the provisions of ISO/IEC JTC 1 Directives.

In accordance with the provisions of Council Resolution 21/1986 this DIS is circulated in the English language only.

Conformément aux dispositions de la Résolution du Conseil 21/1986, ce DIS est distribué en version anglaise seulement.

NOTE FROM ITTF

This draft International Standard is submitted for JTC 1 national body vote under the Fast-Track Procedure.

In accordance with Resolution 30 of the JTC 1 Berlin Plenary 1993, the proposer of this document recommends assignment of ISO/IEC DIS 20970 to JTC 1/SC 22.

See also explanatory report.

"FAST-TRACK" PROCEDURE

1 Any P-member, Category A liaison organization or recognized PAS submitter of ISO/IEC JTC 1, may propose that an existing standard from any source be submitted directly for vote as a DIS. The criteria for proposing an existing standard for the fast-track procedure are a matter for each proposer to decide.

2 The proposal shall be received by the ITTF which will take the following actions.

2.1 To settle the copyright and/or trade mark situation with the proposer, so that the proposed text can be freely copied and distributed within JTC 1 without restriction.

2.2 To assess in consultation with the JTC 1 secretariat which SC is competent for the subject covered by the proposed standard and to ascertain that there is no evident contradiction with other International Standards.

2.3 To distribute the text of the proposed standard as a DIS. In case of particularly bulky documents the ITTF may demand the necessary number of copies from the proposer.

3 The period for combined DIS voting shall be six months. In order to be accepted the DIS must be supported by 75 % of the votes cast (abstention is not counted as a vote) and by two-thirds of the P-members voting of JTC 1.

4 At the end of the voting period, the comments received, whether editorial only or technical, will be dealt with by a working group appointed by the secretariat of the relevant SC.

5 If, after the deliberations of this WG, the requirements of 3 above are met, the amended text shall be sent to the ITTF by the secretariat of the relevant SC for publication as an International Standard.

If it is impossible to agree to a text meeting the above requirements, the proposal has failed and the procedure is terminated.

In either case the WG shall prepare a full report which will be circulated by the ITTF.

6 If the proposed standard is accepted and published, its maintenance will be handled by JTC 1.

JEFF File Format

1 Overview 4
2 Data Types 5
2.1 Basic Types 5
2.2 Language Types 5
2.3 Specific Types 6
2.3.1 Access flags 6
2.3.2 Type Descriptor 7
2.3.3 Offsets 8
2.3.4 Index Values 8
3 File Structure 9
3.1 Definitions 9
3.1.1 Fully Qualified Names 9
3.1.2 Symbolic Names 9
3.1.3 Internal Classes and External Classes 10
3.1.4 Fields and Methods 10
3.1.5 Field Position 10
3.2 Conventions 11
3.2.1 Notations 11
3.2.2 Byte Order 11
3.2.3 Alignment and Padding 12
3.3 The File Structure 12
3.3.1 File Header 13
3.3.2 Class Area 15
3.3.2.1 Class Header 16
3.3.2.2 Interface Table 18
3.3.2.3 Referenced Class Table 18
3.3.2.4 Internal Field Table 18
3.3.2.5 Internal Method Table 19
3.3.2.6 Referenced Field Table 20
3.3.2.7 Referenced Method Table 21
3.3.2.8 Bytecode Block Structure 22
3.3.2.9 Caught Exception Table 23
3.3.2.10 Constant Data Section 23
3.3.3 Attribute Section 25
3.3.3.1 Attribute Type 26
3.3.3.2 Class Attributes 27
3.3.3.3 Attribute Table 27
3.3.4 Symbolic Data Section 28
3.3.5 Constant Data Pool 29
3.3.5.1 Constant Data Pool Structure 29
3.3.5.2 Descriptor 30
3.3.5.3 Method Descriptor 30
3.3.6 File Signature 30
4 Bytecodes 31

4.1 Principles 31
4.2 Translations.. 31
4.2.1 The tableswitch Opcode 32
4.2.2 The lookupswitch Opcode.. 32
4.2.3 The new Opcode 33
4.2.4 Opcodes With Class Arguments 33
4.2.5 The newarray Opcode 34
4.2.6 The multianewarray Opcode.. 34
4.2.7 Field Opcodes 35
4.2.8 Method Opcodes 35
4.2.9 The ldc Opcodes 36
4.2.10 The wide <opcode> Opcodes 37
4.2.11 The wide iinc Opcode 37
4.2.12 Jump Opcodes 38
4.2.13 Long Jump Opcodes.. 38
4.2.14 The sipush Opcode.. 39
4.2.15 The newconstarray Opcode 39
4.3 Unchanged Instructions 40
4.3.1 One-Byte Instructions 40
4.3.2 Two-bytes Instructions 43
4.4 Complete Opcode Mnemonics by Opcode.. 43
5 Restriction 46

1 Overview

This document describes the JEFF File Format. This format is designed to download and store classes on a platform. The distribution of applications is not the target of this specification.

JEFF is a translation of the Java class file format defined in the "Virtual Machine Specification" (see [1]). The class file format defined in [1] is not a ready-for-execution format, i.e. the information stored in this format must be processed further before it can be used for execution by the virtual machine. Consequently, at runtime, the classes stored in this class file format have to be recopied into dynamic runtime memory where they are put in a new form appropriate for execution. Thus, the size of the dynamic runtime memory needed to execute a Java program stored in this class file format is proportional to the size of the stored program. The goal of JEFF is to provide a ready-for-execution format allowing Java programs to be executed directly from static memory, thus avoiding the necessity to recopy classes into dynamic runtime memory for execution.

The constraints put on the design of JEFF are the following:

- Any set of class files must be translatable into a single JEFF file.
- JEFF must be a ready-for-execution format. A virtual machine can use it efficiently, directly from static memory (ROM, flash memory…). No copy in dynamic runtime memory or extra data modification shall be needed.
- All the standard behaviors and features of a Java virtual machine as defined in [1] must be reproducible using JEFF.
- In particular, JEFF must facilitate "symbolic linking" of classes (see [1]). The replacement of a class definition by another class definition having a compatible signature (same class name, same fields and same method signatures) must not require any modifications in the other class definitions.

The main consequences of these choices are:

- A JEFF file can contain several classes from several packages. The content can be a complete application, parts of it, or only one class.
- To allow "symbolic linking" of classes, the references between classes must be kept at the symbolic level, even within a single JEFF file.
- The binary content of a JEFF file is adapted to be efficiently read by a wide range of processors (with different byte orders, alignments, etc.).
- JEFF is also a highly efficient format for the dynamic downloading of class definitions to dynamic memory (RAM).

This specification is a self-contained normative definition of the JEFF format. However, to fully understand the content of this specification, the reading of the following documents is recommended:

[1] The Java™ Virtual Machine Specification, Second Edition, by Tim Lindholm and Franck Yellin, 496 pages, Addison Wesley, April 1999, ISBN 0201432943.
[2] The Java™ Language Specification, Second Edition, by Bill Joy, Guy Steele, James Gosling and Gilad Bracha, 544 pages, Addison Wesley, June 5 2000, ISBN 0201310082.

The limitations introduced by the use of JEFF are described in chapter 5 Restriction.

2 Data Types

This chapter describes the data types used by the JEFF format specification. All the values in a JEFF file are stored on one, two, four or eight bytes. In this document, the expression “null value” is synonym of a value of zero.

2.1 Basic Types

The types **TU1**, **TU2**, and **TU4** represent an unsigned one-, two-, or four-byte quantity, respectively. The types **TS1**, **TS2**, and **TS4** represent a signed one-, two-, four-byte quantity, respectively.

2.2 Language Types

The language types like **int**, **short** or **char** are represented internally as follows:

Format Types	Language Types	Format	Min. Value	Max. Value
JBYTE	**byte**	8-bit signed integer	-128	127
JSHORT	**short**	16-bit signed integer	-32768	32767
JINT	**int**	32-bit signed integer	-2147483648	2147483647
JLONG	**long**	64-bit signed integer	-9.2233e+18	9.2233e+18
JFLOAT	**float**	32-bit IEEE 754	-	
JDOUBLE	**double**	64-bit IEEE 754	-	
JCHAR	**char**	16-bit Unicode char	0	Unicode m[illegible]

Note: The floating-point data are always stored in the file using the **JFLOAT** and **JDOUBLE** format corresponding to 32- and 64-bit IEEE 754 specification. The byte order used is the global byte order used for the whole file. If a specific processor does not use this order, the virtual machine is responsible for the data translation during the download or at runtime.

The character strings are stored in the following structure:

```
VMConstUtf8 {
    TU2 nStringLength;
    TU1 nStringValue[];
}
```

The items of the **VMConstUtf8** structure are as follows:

nStringLength
The length of the encoded string, in bytes. This value may be different from the number of characters in the string.

nStringValue
The string value encoded with the Utf8 format as defined in the Virtual Machine Specification (see [1]).

2.3 Specific Types

Theses types are used to store values with a specific meaning.

Types	Description	Format
VMACCESS	Access Flag (see values below)	16-bit vector
VMTYPE	Type descriptor (see values below)	8-bit vector
VMNCELL	Number of virtual machine cells	16-bit unsigned integer
VMOFFSET	Memory offset (in bytes)	16-bit unsigned integer
VMDOFFSET	Memory offset (in bytes)	32-bit unsigned integer
VMCINDEX	Class Index	16-bit unsigned integer
VMPINDEX	Package Index	16-bit unsigned integer
VMFINDEX	Field Index	32-bit unsigned integer
VMMINDEX	Method Index	32-bit unsigned integer

2.3.1 Access flags

The **VMACCESS** type describes the access privileges for classes, methods and fields. This type is conforming to the access flag type defined in the "Virtual Machine Specification" (see [1]). It's a bit vector with the following values:

Flag Name	Value	Meaning
	Class	
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package.
ACC_FINAL	0x0010	Is final; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods especially in invokespecial.
ACC_INTERFACE	0x0200	Is an interface.
ACC_ABSTRACT	0x0400	Is abstract; may not be instantiated.
	Field	
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Is private; usable only within the defined class.
ACC_PROTECTED	0x0004	Is protected; may be accessed within subclasses.
ACC_STATIC	0x0008	Is static.
ACC_FINAL	0x0010	Is final; no further overriding or assignment after initialization.
ACC_VOLATILE	0x0040	Is volatile; cannot be cached.
ACC_TRANSIENT	0x0080	Is transient; not written or read by a persistent object manager.
	Method	
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Is private; usable only within the defined class.
ACC_PROTECTED	0x0004	Is protected; may be accessed within subclasses.
ACC_STATIC	0x0008	Is static.
ACC_FINAL	0x0010	Is final; no overriding is allowed.
ACC_SYNCHRONIZED	0x0020	Is synchronized; wrap use in monitor lock.
ACC_NATIVE	0x0100	Is native; implemented in a language other than the source language.
ACC_ABSTRACT	0x0400	Is abstract; no implementation is provided.
ACC_STRICT	0x0800	The VM is required to perform strict floating-point

		operations.

2.3.2 Type Descriptor

A type descriptor is composed of a type value (a **VMTYPE**), an optional array dimension value (a **TU1**) and an optional class index (a **VMCINDEX**).

Type Value

The **VMTYPE** type is a byte built with one of the following values:

VM_TYPE_VOID	0x00	Used for the return type of a method
VM_TYPE_SHORT	0x01	
VM_TYPE_INT	0x02	
VM_TYPE_LONG	0x03	
VM_TYPE_BYTE	0x04	
VM_TYPE_CHAR	0x05	
VM_TYPE_FLOAT	0x06	
VM_TYPE_DOUBLE	0x07	
VM_TYPE_BOOLEAN	0x08	
VM_TYPE_OBJECT	0x0A	

These values can be interpreted as a bit field as follows:

```
7—---4 3--2 1--0
 0000 | XX | YY |
```

Where:

- **YY** is the type size in bytes. The size is: 1 << YY
- **XX** is just used to differentiate the types having the same size.

The following flags are also set:

VM_TYPE_TWO_CELL	0x10	for a type using two virtual machine cells (this flag is not set for an array)
VM_TYPE_REF	0x20	for an object or an array
VM_TYPE_MONO	0x40	for a mono-dimensional array
VM_TYPE_MULTI	0x80	for a n-dimensional array, where n >= 2

Dimension Value

The dimension value gives the number of dimensions (0-255) of an array type. This value is optional for a non-array type or for a mono-dimensional array. For a multi-dimensional array, the **VM_TYPE_MULTI** flag is set in the type value and the dimension value is mandatory to know the exact array type.

The dimension values are as follows:

0 for a non-array type,
1 for a simple array (ex: int a[2]),
2 for a 2 dimensional array (ex: long array[2][8]),
...
255 for a 255 dimensional array.

Class Index

The optional class index gives the exact type of descriptor of a class or of an array of class. For a scalar type or an array of scalar types, the class index is useless.

Examples

A simple instance of "String": type = 0x2A, optional dimension = 0x00, class index = index of "java.lang.String"

A primitive type descriptor of a "short": type = 0x01, optional dimension = 0x00, no class index

A simple array of integers (e.g. int[5]): type = 0x62, optional dimension = 0x01, no class index

A simple array of class "MyClass" (e.g. MyClass[5]) : type = 0x6A, optional dimension = 0x01, class index = index of "MyClass"

A primitive type descriptor of a "long": type = 0x13, optional dimension = 0x00, no class index

A 3-dimensional array of long (e.g. long[5][4][]): type = 0xA3, dimension = 0x03, no class index

A 4-dimensional array of class "MyClass" (e.g. MyClass[5][4][][]): type = 0xAA, dimension = 0x04, class index = index of "MyClass"

A "void" return type (for a method): type = 0x00, no dimension, no class index

2.3.3 Offsets

There are two types of offset values used in the specification: **VMOFFSET** and **VMDOFFSET**.

A **VMOFFSET** is an unsigned 16-bit value. This value is an offset in bytes from the beginning of a class file header. Depending of where the offset value is located, the corresponding class file header is unambiguous.

A **VMDOFFSET** is an unsigned 32-bit value. This value is an offset in bytes from the beginning of the file header.

2.3.4 Index Values

See the File Structure.

3 File Structure

This chapter gives the complete structure of the JEFF file format.

3.1 Definitions

This part describes the definitions and rules used in the specification.

3.1.1 Fully Qualified Names

Fully qualified names have the following definition:

- The fully qualified name of a named package that is not a sub-package of a named package is its simple name.
- The fully qualified name of a named package that is a sub-package of another named package consists of the fully qualified name of the containing package followed by the character "/" (Unicode 0x002F) followed by the simple (member) name of the sub-package.
- The fully qualified name of a class or interface that is declared in an unnamed package is the simple name of the class or interface.
- The fully qualified name of a class or interface that is declared in a named package consists of the fully qualified name of the package followed by the character "/" (Unicode 0x002F) followed by the simple name of the class or interface.

3.1.2 Symbolic Names

The file specification refers to symbolic names for the classes, the packages, the fields and the methods. They are defined as follow:

Class Symbolic Name

A class symbolic name is the fully qualified name of the class (package and class names, e.g. "*java/lang/String*"). If a class has no package, the class symbolic name is the class name.

Package Symbolic Name

A package symbolic name is the fully qualified name of the package (e.g. "*java/lang* ").

Field Symbolic Name

A field symbolic name is the concatenation of the field name, a space character (Unicode 0x0020) and the field descriptor string.

e.g. for the field **double m_Field[]**, the symbolic name is "*m_Field [D*".

Method Symbolic Name

A method symbolic name is the concatenation of the method name, a space character (Unicode 0x0020) and the method descriptor string.

e.g. for the method **void append(String)**, the symbolic name is "*append (Ljava/lang/String;)V*".

3.1.3 Internal Classes and External Classes

A JEFF file contains the definition of one or several classes. For a given file, the classes stored in the file are called “internal classes”. The classes referenced by the internal classes but not included in the same file are called “external classes”.

The packages of the internal and external classes are ordered following the crescent lexicographic order of their fully qualified names. This order defines an index value for each package (a **VMPINDEX** value). The package index range is **0** to **number of packages – 1**. If an internal or an external class has no package, this class is defined in the “default packqage”, a package with no name. In this case the “default package” must be counted in the **number of packages** and its index is always 0.

The internal classes and the external classes are ordered and identified by an index (a **VMCINDEX** value). The index range is:

0	to	**InternalClassCount – 1**	for the internal classes
InternalClassCount	to	**TotalClassCount – 1**	for the external classes

The class index values follow the crescent lexicographic order of the class fully qualified names (separately for the internal classes and for the external classes)

The package index and the class index assignments are local to the file.

3.1.4 Fields and Methods

The field indexes are built as follows: The symbolic name of the internal class fields and the symbolic name of the external class fields are ordered in a table following the crescent lexicographic order. The redundancies are eliminated. All the symbolic names representing the internal class fields are stored at the beginning of the table. Each entry in the table is identified by a zero-based index (a **VMFINDEX** value).

By definition of the field symbolic name and the construction of the table, the following properties are deducted:

- Two different field indexes identify two different symbolic names.
- Two different fields, internal or external, share the same index if and only if they have the same name and the same descriptor.

The same construction is used to define the method indexes (**VMMINDEX**).

By definition of the method symbolic name and the construction of the table, the following properties are deducted:

- Two different method indexes identify two different symbolic names.
- Two different methods, internal or external, share the same index if and only if they have the same name and the same descriptor.

The field index and the method index assignments are local to the file.

3.1.5 Field Position

JEFF includes some information about the position of the field in memory. These pre-computed values are useful to speedup the download of classes and to have a quick access to the fields at runtime.

The computation must take into account the following constraints:

- Class fields and instance fields are stored in separate memory spaces.

- The field data must be aligned in memory according to their sizes.
- Most of the virtual machines store the field values contiguously for each class.
- When a class A inherits from a class B, the way the instance fields of an instance of A are stored depends of the virtual machine. Some virtual machines store the fields of A first and then the fields of B, others use the opposite order and other stores them in non-contiguous memory areas.
- The binary compatibility requirement (see Overview) implies that the values computed for a class are independent of the values computed for its super classes, whether or not they are included in the same file.

The consequences of these constraints are the following:

- The pre-computed values are redundant with the field information. They are only included to speedup the virtual machine.
- Some virtual machines may not use these values.
- The values are computed independently for each class.

The same construction process is applied separately for the class fields and the instance fields. The super class fields and the sub-class fields are not taken into account.

- The fields are classed in an ordered list. The order used follows the size of each field. The longer fields are stored first (type long or double), the smaller fields are stored at the end of the list (type byte). The order used between fields of the same size is undefined. This ordering allows keeping the alignment between the data.
- The position of a given field is the position of the preceding field in the list plus the size of the preceding field. The first field position is zero.
- The total size of the field area is the sum of the size of each field in the list.

3.2 Conventions

The following conventions are use in this chapter.

3.2.1 Notations

The format is presented using pseudo-structures written in a C-like structure notation. Like the fields of a C structure, successive items are stored sequentially, with padding and alignment.

3.2.2 Byte Order

All the values are stored using the byte order defined by a set of flags specified in the file header. Floating-point numbers and integer values are treated separately.

3.2.3 Alignment and Padding

If a platform requires [illegible] alignment of the multi-byte values in memory, JEFF allows a[illegible] efficient access to all [illegible] data without byte-by-byte reading.

When a JEFF file is stored on the platform, the first byte of the file header must always be aligned in memory on a 8-byte boundary.

All the items constituting the file are aligned in memory. The following table gives the memory alignment:

Elements	Element size, in bytes	Alignment on memory boundaries of
TU1, TS1, JBYTE, VMTYPE	1	1 byte
TU2, TS2, JSHORT, JCHAR, VMACCESS, VMNCELL, VMOFFSET, VMCINDEX	2	2 bytes
TU4, TS4, JINT, JFLOAT, VMDOFFSET, VMMINDEX, VMFINDEX	4	4 bytes
JLONG, JDOUBLE	8	8 bytes

When aligning data, some extra bytes may be needed for padding. These bytes must be set to null.

Structures are always aligned following the alignment of their first element.

Example:

```
VMStructure {
    VMOFFSET ofAnOffset;
    TU4      nAnyValue;
}
```

The structure is aligned on a 2-byte boundary because **VMOFFSET** is a 2-byte type. The field **nAnyValue** is aligned on a 4-byte boundary. A padding of 2 bytes may be inserted between **ofAnOffset** and **nAnyValue**.

3.3 The File Structure

All the structures defined in this specification are stored in the JEFF file one after the other without overlapping and without any intermediate data other than padding bytes required for alignment. Every unspecified data may be stored in an optional attribute as defined in 3.3.3 Attribute Section.

The file structure is composed of six ordered sections.

Section	Description
File Header	File identification and directory
Class Section	List of class areas
Optional Attributes Section	List of the optional attributes
Symbolic Data Section	The symbolic information used by the classes
Constant Data Pool	Set of common constant data
Digital Signature	Signature of the complete file

File Header
The file header contains the information used to identify the file and a directory to access to the other sections content.

Class Section
The class section describes the content of each class (inheritance, fields, methods and code).

Optional Attributes Section
This optional section contains the optional attributes for the file, the classes, the methods and the fields.

Symbolic Data Section
In this area are stored all the symbolic information used to identify the classes, the methods and the fields.

Constant Data Pool
The constant strings and the descriptors used by the Optional Attribute Section and the Symbolic Data Section are stored in this structure.

Digital Signature
This part contains the digital signature of the complete file.

3.3.1 File Header

The file header is always located at the beginning of the file. In the file structure, some sections have a variable length. The file header contains a directory providing a quick access to these sections.

```
VMFileHeader {
   TU1       nMagicWord1;
   TU1       nMagicWord2;
   TU1       nMagicWord3;
   TU1       nMagicWord4;
   TU1       nFormatVersionMajor;
   TU1       nFormatVersionMinor;
   TU1       nByteOrder;
   TU1       nOptions;
   TU4       nFileLength;
   TU2       nFileVersion;
   TU2       nTotalPackageCount;
   TU2       nInternalClassCount;
   TU2       nTotalClassCount;
   TU4       nTotalFieldCount;
   TU4       nTotalMethodCount;
   VMDOFFSET dofAttributeSection;
   VMDOFFSET dofSymbolicData;
   VMDOFFSET dofConstantDataPool;
   VMDOFFSET dofFileSignature;
   VMDOFFSET dofClassHeader[nInternalClassCount];
}
```

The items of the **VMFileHeader** structure are as follows:

nMagicWord1, nMagicWord2, nMagicWord3, nMagicWord4
The format magic word is **nMagicWord1** = 0x4A, **nMagicWord2** = 0x45, **nMagicWord3** = 0x46 and **nMagicWord4** = 0x46 ("JEFF" in Ascii).

nFormatVersionMajor, nFormatVersionMinor,
Version number of the file format. For this version (1.0), the values are **nFormatVersionMajor** = 0x01 for the major version number and **nFormatVersionMinor** = 0x00 for the minor version number.

nByteOrder
This 8-bit vector gives the byte order used by all the values stored in the file, except the magic number. The following set of flags gives the byte order of integer values and the floating-point values separately. In the definitions, the term “integer value” designs all the two-, four- and height-bytes long values, except the **JFLOAT** and **JDOUBLE** values.

VM_ORDER_INT_BIG	0x01	If this flag is set, integer values are stored using the big-endian convention. Otherwise, they are stored using the little-endian convention.
VM_ORDER_INT_64_INV	0x02	If this flag is set, the two 32-bit parts of the 64-bit integer values are inverted.
VM_ORDER_FLOAT_BIG	0x04	If this flag is set, **JFLOAT** and **JDOUBLE** values are stored using the big-endian convention. Otherwise, they are stored using the little-endian convention.
VM_ORDER_FLOAT_64_INV	0x08	If this flag is set, the two 32-bit parts of the **JDOUBLE** values are inverted.

nOptions
A set of information on the content of the internal classes.

This item is an 8-bit vector with the following flag values:

VM_USE_LONG_TYPE	0x01	One of the classes uses the "**long**" type (in the fields types, the methods signatures, the constant values or the bytecode instructions).
VM_USE_UNICODE	0x02	This file contains non-ASCII characters (Unicode).
VM_USE_FLOAT_TYPE	0x04	One of the classes uses the "**float**" type and/or the "**double**" type (in the fields types, the methods signatures, the constant values or the bytecode instructions).
VM_USE_STRICT_FLOAT	0x08	One of the classes contains bytecodes with strict floating-point computation (the "**strictfp**" keyword is used in the source file).
VM_USE_NATIVE_METHOD	0x10	One of the classes contains native methods.
VM_USE_FINALIZER	0x20	One of the classes has an instance finalizer or a class finalizer.
VM_USE_MONITOR	0x40	One of the classes uses the flag **ACC_SYNCHRONIZED** or the bytecodes **monitorenter** or **monitorexit** in one of its methods.

nFileLength
Size in bytes of the file (all elements included).

nFileVersion
Version number of the file itself. The most significant byte carries the major version number. The less significant byte carries the minor version number. This specification does not define the interpretation of this field by a virtual machine.

nTotalPackageCount
The total number of unique packages referenced in the file (for the internal classes and the external classes).

nInternalClassCount
The number of classes in the file (internal classes).

nTotalClassCount
The total number of the classes referenced in the file (internal classes and external classes).

nTotalFieldCount
The total number of field symbolic names used in the file.

nTotalMethodCount
The total number of method symbolic names used in the file.

dofAttributeSection
Offset of the Optional Attribute Section, a **VMAttributeSection** structure. This field is set to null if no optional attributes are stored in the file.

dofSymbolicData
Offset of the symbolic data section, a **VMSymbolicDataSection** structure.

dofConstantDataPool
Offset of the constant data pool, a **VMConstantDataPool** structure.

dofFileSignature
Offset of the file signature defined in a **VMFileSignature** structure. This value is set to null if the file is not signed.

dofClassHeader
Offsets of the **VMClassHeader** structures for all internal classes. The entries of this table follow the class index order and the class areas are stored in the same order.

3.3.2 Class Area

For each class included in the file, a class area contains the information specific to the class. Within the class area, the references to other elements are given by 16-bit unsigned offsets (**VMOFFSET**) relative to the beginning of the class header.

The first element of this area is the class header pointed to from the **dofClassHeader** array in the file header. The other structures in the class area are stored one after the other without overlapping and without any intermediate data other than padding bytes required for alignment.

The ten sections of the class area must be ordered as follows:

Section	Description
Class Header	Class identification and directory
Interface Table	List of the interfaces implemented by the current class
Referenced Class Table	List of the classes referenced by the current class
Internal Field Table	List of the fields of the current class
Internal Method Table	List of the methods of the current class
Referenced Field Table	List of the fields of other classes used by the current class
Referenced Method Table	List of the methods of other classes used by the current class
Bytecode Area List	List of the bytecode areas for the methods of the current class
Exception Table List	List of the exception handler tables for the methods of the current class
Constant Data Section	Set of constant data used by the current class

3.3.2.1 Class Header

The class header is always located at the beginning of the class representation. In the class file structure, some sections have a variable length. The directory is used as a redirector to have a quick access to these sections.

```
VMClassHeader {
    VMOFFSET   ofThisClassIndex;
    VMPINDEX   pidPackage;
    VMACCESS   aAccessFlag;
    TU2        nClassData;
    VMOFFSET   ofClassConstructor;

    VMOFFSET   ofInterfaceTable;
    VMOFFSET   ofFieldTable;
    VMOFFSET   ofMethodTable;
    VMOFFSET   ofReferencedFieldTable;
    VMOFFSET   ofReferencedMethodTable;
    VMOFFSET   ofReferencedClassTable;
    VMOFFSET   ofConstantDataSection;

    VMOFFSET   ofSuperClassIndex;
    TU2        nInstanceData;
    VMOFFSET   ofInstanceConstructor;
}
```

The items of the **VMClassHeader** structure are as follows:

ofThisClassIndex
Offset of the current class index, a **VMCINDEX** value stored in the “referenced class table” of the current class.

pidPackage
The current class package index.

aAccessFlag
Class access flags. The possible values are:

ACC_PUBLIC	Is public; may be accessed from outside its package.
ACC_FINAL	Is final; no subclasses allowed.

ACC_SUPER	Treat superclass methods specially in invokespecial.
ACC_INTERFACE	Is an interface.
ACC_ABSTRACT	Is abstract; may not be instantiated.

nClassData
This value is the total size, in bytes, of the class fields. The algorithm used to compute the value is given in 3.1.5 Field Position. The size is null if there is no class field in the class.

ofClassConstructor
Offset of the class constructor "**<clinit>**". Offset of the corresponding **VMMethodInfo** structure. Null if there is no class constructor.

ofInterfaceTable
Offset of the interface table, a **VMInterfaceTable** structure. This value is null if the current class implements no interfaces.

ofFieldTable
Offset of the internal field table, a **VMFieldInfoTable** structure. This value is null if the current class has no field.

ofMethodTable
Offset of the internal method table, a **VMMethodInfoTable** structure. This value is null if the current class has no method.

ofReferencedFieldTable
Offset of the referenced field table, a **VMReferencedFieldTable** structure. This value is null if the bytecode uses no field.

ofReferencedMethodTable
Offset of the referenced method table, a **VMReferencedMethodTable** structure. This value is null if the bytecode uses no method.

ofReferencedClassTable
Offset of the referenced class table, a **VMReferencedClassTable** structure.

ofConstantDataSection
Offset of the constant data section, a **VMConstantDataSection** structure. This value is null if the class does not contain any constants.

ofSuperClassIndex
Offset of the super class index, a **VMCINDEX** value stored in the “referenced class table” of the current class. If the current class is **java.lang.Object,** the offset value is zero. This value is not present for an interface.

nInstanceData
This value is the total size, in bytes, of the instance fields. The algorithm used to compute the value is given in 3.1.5 Field Position. The size is null if there is no instance field in the class. This value is not present for an interface

ofInstanceConstructor
Offset of the default instance constructor "**<init> ()V**". Offset of the corresponding **VMMethodInfo** structure. The value is null if there is no default instance constructor. This value is not present for an interface.

3.3.2.2 Interface Table

This structure is the list of the interfaces implemented by this class or interface.

```
VMInterfaceTable {
    TU2      nInterfaceCount;
    VMOFFSET ofInterfaceIndex [nInterfaceCount];
}
```

The items of the **VMInterfaceTable** structure are as follows:

nInterfaceCount
The number of interfaces implemented.

ofInterfaceIndex
Offset of a class index, a **VMCINDEX** value stored in the "referenced class table" of the current class. The corresponding class is a super interface implemented by the current class or interface.

3.3.2.3 Referenced Class Table

Every class, internal or external, referenced by the current class is represented in the following table:

```
VMReferencedClassTable {
    TU2 nReferencedClassCount;
    VMCINDEX cidReferencedClass [nReferencedClassCount];
}
```

The current class is also represented in this table.

The items of the **VMReferenceClassTable** structure are as follows:

nReferencedClassCount
The number of referenced classes.

cidReferencedClass
The class index (**VMCINDEX** value) of a class referenced by the current class.

3.3.2.4 Internal Field Table

Every field member of the defined class is described by a field information structure located in a table:

```
VMFieldInfoTable {
    TU2 nFieldCount;
    {
        VMFINDEX   fidFieldIndex;
        VMOFFSET   ofThisClassIndex;
        VMTYPE     tFieldType;
        TU1        nTypeDimension;
        VMACCESS   aAccessFlag;
        TU2        nFieldDataOffset;
    } VMFieldInfo [nFieldCount];
}
```

The instance fields are always stored first in the table. The class fields follow them. Instance fields and class fields are stored following the crescent order of their index. The items of the **VMFieldInfoTable** structure are as follows:

nFieldCount
The number of fields in the class.

fidFieldIndex
The field index.

ofThisClassIndex
Offset of the current class index, a **VMCINDEX** value stored in the "referenced class table" of the current class.

tFieldType
The field type. By definition, the field type gives the size of the value stored by the field.

nTypeDimension
The array dimension associated with the type. This value is always present.

aAccessFlag
Field access flag. The possible values are:

ACC_PUBLIC	Is public; may be accessed from outside its package.
ACC_PRIVATE	Is private; usable only within the defined class.
ACC_PROTECTED	Is protected; may be accessed within subclasses.
ACC_STATIC	Is static.
ACC_FINAL	Is final; no further overriding or assignment after initialization.
ACC_VOLATILE	Is volatile; cannot be cached.
ACC_TRANSIENT	Is transient; not written or read by a persistent object manager.

nFieldDataOffset
This value is an offset, in bytes, of the field data in the class field value area or in the instance value area. The algorithm used to compute the value is given in 3.1.5 Field Position. The total size of the instance field data area is given by **nInstanceData**. The total size of the class field data area is given by **nClassData**.

3.3.2.5 Internal Method Table

Every method of the defined class, including the special internal methods, **<init>** or **<clinit>,** is described by a method information structure located in a table:

```
VMMethodInfoTable {
    TU2 nMethodCount;
    {
        VMMINDEX   midMethodIndex;
        VMOFFSET   ofThisClassIndex;
        VMNCELL    ncStackArgument;
        VMACCESS   aAccessFlag;
        VMOFFSET   ofCode;
    } VMMethodInfo [nMethodCount];

    TU4 nNativeReference[];
}
```

The instance methods are always stored first in the table. The class methods follow them. Instance methods and class methods are stored following the crescent order of their index. The items of the **VMMethodInfoTable** structure are as follows:

nMethodCount
The number of method in the class.

midMethodIndex
The method index.

ofThisClassIndex
Offset of the current class index, a **VMCINDEX** value stored in the “referenced class table” of the current class.

ncStackArgument
Size of the method arguments in the stack. The size includes the reference to the instance used for calling an instance method. This size does not include the return value of the method. The bytecode interpreter uses **ncStackArgument** to clean the stack after the method return. The size, in cells, is computed during the class translation.

aAccessFlag
Method access flag. The possible values are:

ACC_PUBLIC	Is public; may be accessed from outside its package.
ACC_PRIVATE	Is private; usable only within the defined class.
ACC_PROTECTED	Is protected; may be accessed within subclasses.
ACC_STATIC	Is static.
ACC_FINAL	Is final; no overriding is allowed.
ACC_SYNCHRONIZED	Is synchronized; wrap use in monitor lock.
ACC_NATIVE	Is native; implemented in a language other than the source language.
ACC_ABSTRACT	Is abstract; no implementation is provided.
ACC_STRICT	The VM is required to perform strict floating-point operations.

ofCode
For a non-native non-abstract method, this value is the offset of the bytecode block, a **VMBytecodeBlock** structure. For an abstract method, the offset value is null. For a native method, the value is the offset of one of the **nNativeReference** values. Each native method must refer to a separate **nNativeReference** value.

nNativeReference
This array of undefined **TU4** values must contain as many elements as the class has native methods. These values are reserved for future use.

3.3.2.6 Referenced Field Table

The referenced field table describes the internal or external class fields that are not members of the current class but are used by this class. If an instruction refers to such a field, the bytecode gives the offset of the corresponding **VMReferencedField** structure.

```
VMReferencedFieldTable {
    TU2 nFieldCount;
    {
        VMFINDEX  fidFieldIndex;
        VMOFFSET  ofClassIndex;
        VMTYPE    tFieldType;
        TU1       nTypeDimension;
    } VMReferencedField [nFieldCount];
}
```

The items of the **VMReferencedFieldTable** structure are as follows:

nFieldCount
The number of fields in the table.

fidFieldIndex
The field index.

ofClassIndex
Offset of a class index, a **VMCINDEX** value stored in the "referenced class table" of the current class. This index identifies the class containing the field.

tFieldType
The field type. By definition, the field type gives the size of the value stored by the field. This information is used to retrieve in the operand stack the reference of the object instance (for an instance field).

nTypeDimension
The array dimension associated with the type. This value is always present.

3.3.2.7 Referenced Method Table

The referenced method table describes the internal or external class methods that are not members of the current class but are used by this class. If an instruction refers to such a method, the bytecode gives the offset of the corresponding **VMReferencedMethod** structure.

```
VMReferencedMethodTable {
    TU2 nMethodCount;
    {
        VMMINDEX  midMethodIndex;
        VMOFFSET  ofClassIndex;
        VMNCELL   ncStackArgument;
    } VMReferencedMethod [nMethodCount];
}
```

The items of the **VMReferencedMethodTable** structure are as follows:

nMethodCount
The number of methods in the table.

midMethodIndex
The method index.

ofClassIndex
Offset of a class index, a **VMCINDEX** value stored in the "referenced class table" of the current class. This index identifies the class containing the method.

ncStackArgument
Size of the method arguments in the stack. The size includes the reference to the instance used for calling an instance method. This size does not include the return value of the method. The bytecode interpreter uses **ncStackArgument** to clean the stack after the method return. The size, in cells, is computed during the class translation.

3.3.2.8 Bytecode Block Structure

This part is a block of bytecode corresponding to the method body:

```
VMBytecodeBlock {
    VMNCELL   ncMaxStack;
    VMNCELL   ncMaxLocals;
    VMOFFSET  ofExceptionCatchTable;
    TU2       nByteCodeSize;
    TU1       bytecode[nByteCodeSize];
}
```

The items of the **VMBytecodeBlock** structure are as follows:

ncMaxStack
The value of the **ncMaxStack** item gives the maximum number of cells on the operand stack at any point during execution of this method.

ncMaxLocals
The value of the **ncMaxLocals** item gives the number of local variables used by this method, including the parameters passed to the method on invocation. The index of the first local variable is 0. The greatest local variable index for a one-word value is **ncMaxLocals-1**. The greatest local variable index for a two-word value is **ncMaxLocals-2**.

ofExceptionCatchTable
Offset of the caught exception table, a **VMExceptionCatchTable** structure. Null if no exception is caught in this method.

nByteCodeSize
The size of the bytecode block in bytes. The value of **nByteCodeSize** must be greater than zero; the code array must not be empty.

bytecode
The bytecode area contains the instructions for the method. All branching instructions included in a bytecode area must specify addresses within the same bytecode area. All exception handlers defined for a bytecode area must reference addresses within that bytecode area. The bytecode area may only contain bytecodes defined in this specification, their arguments and padding bytes (if needed for alignment).

Note for the class initializer

Since the initialization values of the static fields are not included in JEFF, a piece of code must be added at the beginning of the class initializer “**<clinit>**” to perform the initialization of these fields (if needed).

3.3.2.9 Caught Exception Table

This structure gives the exception handling information for a method. It describes exception handlers semantically equivalent and in the same order as the **exception_table** item of the **Code_attribute** structure defined in the Virtual Machine Specification [1].

```
VMExceptionCatchTable {
    TU2 nCatchCount;
    {
        VMOFFSET  ofStartPc;
        VMOFFSET  ofEndPc;
        VMOFFSET  ofHandlerPc;
        VMOFFSET  ofExceptionIndex;
    } VMExceptionCatch [nCatchCount];
}
```

The items of the **VMExceptionCatchTable** structure are as follows:

nCatchCount
The value of the **nCatchCount** item indicates the number of element in the table.

ofStartPc
Offset of the first byte of the first bytecode in the range where the exception handler is active.

ofEndPc
Offset of the first byte following the last byte of the last bytecode in the range where the exception handler is active.

ofHandlerPc
Offset of the first byte of the first bytecode of the exception handler.

ofExceptionIndex
Offset of a class index, a **VMCINDEX** value stored in the “referenced class table” of the current class. This index identifies the class of the caught exception. The offset value is null if the exception handler has to be called for any kind of exception.

3.3.2.10 Constant Data Section

This section contains the constant data values of the class. They are always referred through an offset.

Single values of type **JINT**, **JLONG**, **JFLOAT** or **JDOUBLE** can be referred by the bytecodes **ildc**, **lldc**, **fldc** and **dldc**. The **VMConstUtf8** structures are referred by the **sldc** bytecode.

The **newconstarray** bytecode refers contiguous set of values of type **JDOUBLE**, **JLONG**, **JFLOAT**, **JINT**, **JSHORT** and **JBYTE**. This bytecode also uses the Utf8 strings stored in **VMConstUtf8** structures to create character arrays.

```
VMConstantDataSection {
    TU2     nConstFlags;
    TU2     nDoubleNumber;
    TU2     nLongNumber;
    TU2     nFloatNumber;
    TU2     nIntNumber;
    TU2     nShortNumber;
    TU2     nByteNumber;
    TU2     nStringNumber;
    JDOUBLE nDoubleValue[nDoubleNumber];
    JLONG   nLongValue[nLongNumber];
    JFLOAT  nFloatValue[nFloatNumber];
    JINT    nIntValue[nIntNumber];
    JSHORT  nShortValue[nShortNumber];
    JBYTE   nByteValue[nByteNumber];
    VMConstUtf8 strConstString[nStringNumber];
}
```

The items of the **VMConstantDataSection** structure are as follows:

nConstFlags
The **nConstFlags** value is a set of flags giving the content of the section as follows:

VM_CONST_DOUBLE	0x0001	The section contains values of type **double**
VM_CONST_LONG	0x0002	The section contains values of type **long**
VM_CONST_FLOAT	0x0004	The section contains values of type **float**
VM_CONST_INT	0x0008	The section contains values of type **int**
VM_CONST_SHORT	0x0010	The section contains values of type **short**
VM_CONST_BYTE	0x0020	The section contains values of type **byte**
VM_CONST_STRING	0x0040	The section contains constant strings

nDoubleNumber
The number of **JDOUBLE** values. This non-null value is only present if the **VM_CONST_DOUBLE** flag is set in **nConstFlags**.

nLongNumber
The number of **JLONG** values. This non-null value is only present if the **VM_CONST_LONG** flag is set in **nConstFlags**.

nFloatNumber
The number of **JFLOAT** values. This non-null value is only present if the **VM_CONST_FLOAT** flag is set in **nConstFlags**.

nIntNumber
The number of **JINT** values. This non-null value is only present if the **VM_CONST_INT** flag is set in **nConstFlags**.

nShortNumber
The number of **JSHORT** values. This non-null value is only present if the **VM_CONST_SHORT** flag is set in **nConstFlags**.

nByteNumber
The number of **JBYTE** values. This non-null value is only present if the **VM_CONST_BYTE** flag is set in **nConstFlags**.

nStringNumber
The number of **VMConstUtf8** structures. This non-null value is only present if the **VM_CONST_STRING** flag is set in **nConstFlags**.

nDoubleValue
A value of type **double**.

nLongValue
A value of type **long**.

nFloatValue
A value of type **float**.

nIntValue
A value of type **int**.

nShortValue
A value of type **short**.

nByteValue
A value of type **byte**.

strConstString
A constant string value (See the definition of the **VMConstUtf8** structure).

nStringValue
The string value encoded with the Utf8 format as defined in the Virtual Machine Specification (see [1]).

3.3.3 Attribute Section

This optional section contains the optional attributes for the file, the classes, the methods and the fields. The format for the translation of the attributes described in the Virtual Machine Specification (see [1]) will be included in an Annex of the JEFF specification.

```
VMAttributeSection {
   VMDOFFSET dofFileAttributeList;
   VMDOFFSET dofClassAttributes[nInternalClassCount];
   TU2       nAttributeTypeCount;
   TU2       nClassAttributeCount;
   VMAttributeType   sAttributeType[nAttributeTypeCount];
   VMClassAttributes sClassAttributes[nClassAttributeCount]
   TU2       nAttributeTableCount;
   VMAttributeTable  sAttributeTable[nAttributeTableCount];
}
```

The **nInternalClassCount** value is defined in the file header.

The items of the **VMAttributeSection** structure are as follows:

dofFileAttributeList
This value is the offset of a **VMAttributeTable** structure. This structure defines the attribute list of the file. The offset value is zero if and only if the JEFF file has no file attributes.

dofClassAttributes
The index in this table is the class index. Each entry value is the offset of a **VMClassAttributes** structure. This structure defines the attributes for the internal class of same index. The offset value iszero if and only if the corresponding class has no attributes.

nAttributeTypeCount
This value is the number of attribute types used in the file.

nClassAttributeCount
This value is the number of **VMClassAttributes** structures used in the file.

nAttributeTableCount
This value is the number of attribute lists (**VMAttributeTable** structures) used in the file.

3.3.3.1 Attribute Type

This structure defines an attribute type.

```
VMAttributeType {
   VMDOFFSET  dofTypeName;
   TU2        nTypeFlags;
   TU2        nTypeLength;
}
```

The items of the **VMAttributeType** structure are as follows:

dofTypeName
Offset of a **VMConstUtf8** structure stored in the constant data pool. The string value is the attribute type name. The type name format follows the rules defined in the Virtual Machine Specification (See [1]).

nTypeFlags
This value is a set of flags defining the attribute type. The flag values are the following:

VM_ATTR_INDEXES	0x0001	The attribute contains some index values of type **VMPINDEX**, **VMCINDEX**, **VMMINDEX** or **VMFINDEX**.
VM_ATTR_VMOFFSETS	0x0002	The attribute contains some values of type **VMOFFSET**.
VM_ATTR_VMDOFFSETS	0x0004	The attribute contains some values of type **VMDOFFSET**.
VM_ATTR_BYTE_ORDER	0x0008	The elements stored in **nData** (See the **VMAttributeTable** structure) contain byte ordered values.
VM_ATTR_CST_LENGTH	0x0010	The length of the attribute is constant and given by the **nTypeLength** item. This flag can only be used if the length of the attribute structure is not subject to variations caused by the type alignment.

nTypeLength
This value is the fixed length of the attribute in bytes, not including the type index (See the **VMAttributeTable** structure). This value is null if the **VM_ATTR_CST_LENGTH** flag is not set in **nTypeFlags**.

3.3.3.2 Class Attributes

The attributes used by a class such as the class attributes, the method attribute and the field attributes are defined in this structure.

```
VMClassAttributes {
   VMDOFFSET  dofClassAttributeList;
   VMDOFFSET  dofFieldAttributeList[nFieldCount];
   VMDOFFSET  dofMethodAttributeList[nMethodCount];
}
```

The items of the **VMClassAttribute** structure are as follows:

dofClassAttributeList
This value is the offset of a **VMAttributeTable** structure. This structure defines the attribute list of the class.

dofFieldAttributeList
This item defines the attribute list of a field. The value is the offset of a **VMAttributeTable** structure. The position of the offset in the list is equal to the position of the field in the internal field list of the corresponding class. The value of the offset is null if the field has no attributes. The value of **nFieldCount** is given by the internal field table structure of the corresponding class.

dofMethodAttributeList
This item defines the attribute list of a method. The value is the offset of a **VMAttributeTable** structure. The position of the offset in the list is equal to the position of the method in the internal method list of the corresponding class. The value of the offset is null if the method has no attributes. The value of **nMethodCount** is given by the internal method table structure of the corresponding class.

3.3.3.3 Attribute Table

This structure is used to store each attribute list.

```
VMAttributeTable {
   TU2 nAttributeCount;
   {
      TU2  nAttributeType;
      TU4  nTypeLength;
      TU1  nData[nTypeLength];
   } VMAttribute[nAttributeCount]
}
```

The items of the **VMAttributeTable** structure are as follows:

nAttributeType
This value is the index of a **VMAttributeType** structure in the attribute type table. The structure defines the type of the attribute.

nTypeLength
This value is the length, in bytes, of the **nData** array. This value is only present if the **VM_ATTR_CST_LENGTH** flag is not set in **nTypeFlags** item of the **VMAttributeType** structure pointed to by **dofAttributeType**. The value must take in account variations of length due to type alignment in the structure of the attribute.

nData
The structure presented is a generic structure that all the attributes must follow. The **nData** byte array stands for the true attribute data.

3.3.4 Symbolic Data Section

This section contains the symbolic information used to identify the elements of the internal and external classes. The reflection feature also uses this section.

```
VMSymbolicDataSection {
   VMPINDEX  pidExtClassPackage[nTotalClassCount-nInternalClassCount];
   VMDOFFSET dofPackageName[nTotalPackageCount];
   VMDOFFSET dofClassName[nTotalClassCount];

   {
      VMDOFFSET  dofFieldName;
      VMDOFFSET  dofFieldDescriptor;
   } VMFieldSymbolicInfo[nTotalFieldCount]

   {
      VMDOFFSET  dofMethodName;
      VMDOFFSET  dofMethodDescriptor;
   } VMMethodSymbolicInfo[nTotalMethodCount]
}
```

The **nTotalPackageCount**, **nTotalClassCount**, **nInternalClassCount**, **nTotalFieldCount** and **nTotalMethodCount** values are defined in the file header.

The items of the **VMSymbolicDataSection** structure are as follows:

pidExtClassPackage
This table gives the package of the corresponding external class. If **n** is a zero-based index in this table, the corresponding entry **pidExtClassPackage[n]**, gives the package index for the external class with a class index value of **n + nInternalClassCount**.

dofPackageName
Offset of a **VMConstUtf8** structure stored in the constant data pool. The string value is the package fully qualified name. The index used in this table is the package index (a **VMPINDEX** value). If the JEFF file references the “default package”, a package with no name, the corresponding **dofPackageName** value is the offset of a **VMConstUtf8** structure with a null length.

dofClassName
Offset of a **VMConstUtf8** structure stored in the constant data pool. The string value is the simple (not fully qualified) class name. The index of an entry in this table is the class index (a **VMCINDEX** value).

VMFieldSymbolicInfo
Table of field symbolic information. The index of an entry in this table is the field index (a **VMFINDEX** value).

dofFieldName
Offset of a **VMConstUtf8** structure stored in the constant data pool. The string value is the simple (not fully qualified) field name.

dofFieldDescriptor
Offset of a **VMDescriptor** structure stored in the constant data pool. The descriptor value gives the field type.

VMMethodSymbolicInfo
Table of method symbolic information. The index of an entry in this table is the method index (a **VMMINDEX** value).

dofMethodName
The value is an offset of a **VMConstUtf8** structure stored in the constant data pool representing either one of the special internal method names, either **<init>** or **<clinit>**, or a method name, stored as a simple (not fully qualified) name.

dofMethodDescriptor
Offset of a **VMMethodDescriptor** structure stored in the constant data pool. The descriptor gives the type of the method arguments and the type of return value.

3.3.5 Constant Data Pool

This structure stores the constant strings and the descriptors used by the Optional Attribute Section and the Symbolic Data Section.

3.3.5.1 Constant Data Pool Structure

```
VMConstantDataPool {
   TU4          nStringCount;
   TU4          nDescriptorCount;
   TU4          nMethodDescriptorCount;
   VMConstUtf8  strConstantString[nStringCount];
   VMDescriptor sDescriptor[nDescriptorCount];
   VMMethodDescriptor sMethodDescriptor[nMethodDescriptorCount];
}
```

The items of the **VMConstantDataPool** structure are as follows:

nStringCount
The number of constant strings stored in the structure.

nDescriptorCount
The number of individual descriptors stored in the structure. This number does not take in account the descriptors included in the method descriptors.

nMethodDescriptorCount
The number of method descriptors stored in the structure.

strConstantString
A constant string value (See the definition of the **VMConstUtf8** structure).

sDescriptor
A descriptor value as defined below.

sMethodDescriptor
A method descriptor value as defined below.

3.3.5.2 Descriptor

```
VMDescriptor
{
    VMTYPE     tDataType;
    TU1        nDataTypeDimension;
    VMCINDEX   cidDataTypeIndex;
}
```

The items of the **VMDescriptor** structure are as follows:

tDataType
The data type. It must be associated to the **nDataTypeDimension** and **cidDataTypeIndex** items to have the full field descriptor.

nDataTypeDimension
The array dimension associated with the type. This value is only present if the type is a n-dimensional array, where n >= 2.

cidDataTypeIndex
The class index associated with the data type. This item is present only if the **tDataType** is not a primitive type or an array of primitive types.

3.3.5.3 Method Descriptor

```
VMMethodDescriptor {
    TU2 nArgCount;
    VMDescriptor sArgumentType[nArgCount];
    VMDescriptor sReturnType;
}
```

The items of the **VMMethodDescriptor** structure are as follows:

nArgCount
The number of argument. 0 for a method without argument.

sArgumentType
The descriptor of an argument type.

sReturnType
The descriptor of the type returned by the method.

3.3.6 File Signature

The **VMFileSignature** structure is not defined.

4 Bytecodes

This chapter describes the instruction set used in JEFF. The operational semantics of the instruction is not provided, as it does not impact the structural description of the JEFF format.

An instruction is an opcode followed by its arguments. An opcode itself is coded on one byte. A <n>-bytes instruction is an instruction of which arguments take <n-1> bytes. A one-byte instruction is an instruction without argument. A two-bytes instruction is an instruction with one argument coded on one byte.

4.1 Principles

The section 4.2 describes only the differences between the class file bytecodes and the JEFF bytecodes. The two instruction sets are equivalent in term of functionalities. The main purpose of the bytecode translation is to create an efficient instruction set adapted to the structure of the file.

Translation Rules

Several operations are applied to the bytecode:

- The replacement. A bytecode is replaced by another bytecode with the same behavior but using another syntax for its arguments.
- The bytecode splitting. A single bytecode with a wide set of functionalities is replaced by several bytecodes implementing a part of the original behavior. The choice of the new bytecode depends on the context.
- The bytecode grouping. A group of bytecodes frequently used is replaced by a new single bytecode performing the same task.

If an instruction is not described in section 4.2, its syntax shall be unchanged with respect to the one assigned to the instruction of same opcode value in class file bytecode (the mnemonic of the opcode is then the mnemonic of the original opcode as found in class file bytecode prefixed by "jeff-").

The instructions of JEFF bytecode that result from a particular translation are completely defined in section 4.2.

All the instructions not described in section 4.2 are one-byte or two-bytes instructions and are defined in section 4.3.

Section 4.4 provides the complete set of opcodes with their mnemonics used in JEFF bytecode.

Alignment and Padding

The bytecodes and their arguments follow the rules of alignment and padding defined in 3.2.3 Alignment and Padding.

4.2 Translations

This chapter defines normatively all the instructions of JEFF bytecode that are not exactly the same than those found in the class file format bytecode. This chapter describes also all the translation operations from which these JEFF instructions result, but this description is not necessary for the intrinsic definition of the JEFF instructions and the references to the instruction set of class file format are here provided only for information purpose.

4.2.1 The tableswitch Opcode

If the original structure of class file bytecode contains the following sequence:

```
TU1 tableswitch
TU1 <0-3 byte pad>
TS4 nDefault
TS4 nLowValue
TS4 nHighValue
TS4 nOffset [nHighValue - nLowValue + 1]
```

Where immediately after the padding follow a series of signed 32-bit values: **nDefault**, **nLowValue**, **nHighValue** and then **nHighValue** - **nLowValue** + 1 further signed 32-bit offsets.

The translated structure shall be the following sequence:

If the **nLowValue** and **nHighValue** values can be converted in 16-bit signed value, the translated structure is:

```
TU1      jeff_stableswitch
TU1      <0-1 byte pad>
VMOFFSET ofDefault
TS2      nLowValue
TS2      nHighValue
VMOFFSET ofJump [nHighValue - nLowValue + 1]
```

Otherwise, the translated structure is:

```
TU1      jeff_tableswitch
TU1      <0-1 byte pad>
VMOFFSET ofDefault
TU1      <0-2 byte pad>
TS4      nLowValue
TS4      nHighValue
VMOFFSET ofJump [nHighValue - nLowValue + 1]
```

The **ofDefault** and **ofJump** values are the jump addresses in the current bytecode block (offsets in bytes from the beginning of the class header structure).

4.2.2 The lookupswitch Opcode

If the original instruction in class file format is:

```
TU1 lookupswitch
TU1 <0-3 byte pad>
TS4 nDefault
TU4 nPairs
      match-offset pairs...
TS4 nMatch
TS4 nOffset
```

Where immediately after the padding follow a series of signed 32-bit values: **nDefault**, **nPairs**, and then **nPairs** pairs of signed 32-bit values. Each of the **nPairs** pairs consists of an **int nMatch** and a signed 32-bit **nOffset**.

The translated structure shall be the following sequence:

If all of the **nMatch** values can be converted in 16-bit signed value, the translated structure is:

```
TU1      jeff_slookupswitch
TU1      <0-1 byte pad>
VMOFFSET ofDefault
TU2      nPairs
TS2      nMatch [nPairs]
VMOFFSET ofJump [nPairs]
```

Otherwise, the translated structure is:

```
TU1      jeff_lookupswitch
TU1      <0-1 byte pad>
VMOFFSET ofDefault
TU2      nPairs
TU1      <0-2 byte pad>
TS4      nMatch [nPairs]
VMOFFSET ofJump [nPairs]
```

The **ofDefault** and **ofJump** values are the jump addresses in the current bytecode block (offsets in bytes from the beginning of the class header structure).

4.2.3 The new Opcode

If the original instruction in class file format is:

```
TU1 new
TU2 nIndex
```

Where the **nIndex** value is an index into the constant pool of the local class. The constant pool entry at this index is a **CONSTANT_Class**.

The translated structure shall be the following sequence:

```
TU1      jeff_new
TU1      <0-1 byte pad>
VMOFFSET ofClassIndex
```

Where the **ofClassIndex** value is the offset of the class index, a **VMCINDEX** value stored in the “referenced class table” of the current class.

4.2.4 Opcodes With Class Arguments

If the original instruction in class file format is:

```
TU1 <opcode>
TU2 nIndex
```

Where **<opcode>** is **anewarray**, **checkcast** or **instanceof**. The **nIndex** value is an index into the constant pool of the local class. The constant pool entry at this index is a **CONSTANT_Class**.

The translated structure shall be a variable-length instruction:

```
TU1      <jeff_opcode>
VMTYPE   tDescriptor
TU1      nDimension    (optional)
TU1      <0-1 byte pad>
VMOFFSET ofClassIndex  (optional)
```

The opcode translation array is:

classfile opcode	jeff opcode
anewarray	jeff_newarray
checkcast	jeff_checkcast
instanceof	jeff_instanceof

The **tDescriptor** value reflects the **CONSTANT_Class** information. The descriptor associated with the **jeff_newarray** bytecode has an array dimension equal to the array dimension of **CONSTANT_Class** structure plus one. The **nDimension** value is the array dimension associated with the descriptor. This value is only present if the **VM_TYPE_MULTI** is set in the **tDescriptor** value. The **ofClassIndex** value is only present if **tDescriptor** describes a class or an array of classes. It's the offset of the class index, a **VMCINDEX** value stored in the “referenced class table” of the current class.

4.2.5 The newarray Opcode

If the original instruction in class file format is:

```
TU1 newarray
TU1 nType
```

Where the **nType** is a code that indicates the type of array to create.

The translated structure shall be the following sequence:

```
TU1    jeff_newarray
VMTYPE tDescriptor
```

The **tDescriptor** value reflects the **nType** information. The **VM_TYPE_MONO** flag is always set in this value.

4.2.6 The multianewarray Opcode

If the original instruction in class file format is:

```
TU1 multianewarray
TU2 nIndex
TU1 nDimensions
```

Where the **nIndex** value is an index into the constant pool of the local class. The constant pool entry at this index is a **CONSTANT_Class**. The **nDimensions** value represents the number of dimensions of the array to be created.

The translated structure shall be a variable-length instruction:

```
TU1      jeff_multianewarray
TU1      nDimensions
VMTYPE   tDescriptor
TU1      nArrayDimension
TU1      <0-1 byte pad>
VMOFFSET ofClassIndex  (optional)
```

The **tDescriptor** value reflects the **CONSTANT_Class** information. The **nArrayDimension** value is the array dimension associated with the descriptor. This value is only present if the **VM_TYPE_MULTI** is set in the **tDescriptor** value. The **ofClassIndex** value is only present if **tDescriptor** describes a class or an array of classes. It's the offset of the class index, a **VMCINDEX** value stored in the "referenced class table" of the current class.

4.2.7 Field Opcodes

If the original instruction in class file format is:

```
TU1 <opcode>
TU2 nIndex
```

Where **<opcode>** is **getfield**, **getstatic**, **putfield** or **putstatic**. The **nIndex** value is an index into the constant pool of the local class. The constant pool entry at this index is a **CONSTANT_Fieldref**.

The translated structure shall be the following sequence:

```
TU1      <jeff opcode>
TU1      <0-1 byte pad>
VMOFFSET ofFieldInfo
```

The opcode translation array is:

classfile opcode	jeff opcode
getfield	jeff_getfield
getstatic	jeff_getstatic
putfield	jeff_putfield
putstatic	jeff_putstatic

If the instruction points to a field of the current class, the **ofFieldInfo** value is the offset of a **VMFieldInfo** structure in the field list of the current class. If the field belongs to another class, the value of **ofFieldInfo** is the offset of a **VMReferencedField** structure in the "referenced field table" of the current class.

4.2.8 Method Opcodes

If the original instruction in class file format is:

```
TU1 <opcode>
TU2 nIndex
```

Where **<opcode>** is **invokespecial**, **invokevirtual**, or **invokestatic**. The **nIndex** value is an index into the constant pool of the local class. The constant pool entry at this index is a **CONSTANT_Methodref** structure.

or

```
TU1 invokeinterface
TU2 nIndex
TU1 nArgs
TU1 0
```

Where the **nIndex** value is an index into the constant pool of the local class. The constant pool entry at this index is a **CONSTANT_InterfaceMethodref** structure. The **nArgs** value is the size in words of the method's arguments in the stack.

The translated structure shall be the following sequence:

```
TU1      <jeff opcode>
TU1      <0-1 byte pad>
VMOFFSET ofMethodInfo
```

The opcode translation array is:

classfile opcode	**jeff opcode**
invokespecial	jeff_invokespecial
invokevirtual	jeff_invokevirtual
invokestatic	jeff_invokestatic
invokeinterface	jeff_invokeinterface

If the instruction points to a method of the current class, the **ofMethodInfo** value is the offset of a **VMMethodInfo** structure in the method list of the current class. If the method belongs to another class, the value of **ofMethodInfo** is the offset of a **VMReferencedMethod** structure in the “referenced method table” of the current class.

4.2.9 The ldc Opcodes

If the original instruction in class file format is:

```
TU1 ldc
TU1 nIndex
```

or

```
TU1 ldc_w
TU2 nIndex
```

Where the **nIndex** value is an index into the constant pool of the local class. The constant pool entry at this index is a **CONSTANT_Integer**, a **CONSTANT_Float**, or a **CONSTANT_String**.

or

```
TU1 ldc2_w
TU2 nIndex
```

Where the **nIndex** value is an index into the constant pool of the local class. The constant pool entry at this index is a **CONSTANT_Long**, or a **CONSTANT_Double**.

The translated structure shall be the following sequence:

```
TU1      <jeff opcode>
TU1      <0-1 byte pad>
VMOFFSET ofConstant
```

Where **<jeff opcode>** depends of the constant type. The **ofConstant** value is the offset of a data value stored in the constant data section. The type of the value depends of the constant type.

constant type	jeff opcode	type of the value pointed to by ofConstant
CONSTANT_String	jeff_sldc	VMConstUtf8
CONSTANT_Integer	jeff_ildc	JINT
CONSTANT_Float	jeff_fldc	JFLOAT
CONSTANT_Long	jeff_lldc	JLONG
CONSTANT_Double	jeff_dldc	JDOUBLE

4.2.10 The wide <opcode> Opcodes

If the original instruction in class file format is:

```
TU1 wide
TU1 <opcode>
TU2 nIndex
```

Where **<opcode>** is **aload, astore, dload, dstore, fload, fstore, iload, istore, lload, lstore**, or **ret**. The **nIndex** value is an index to a local variable in the current frame.

The translated structure shall be the following sequence:

```
TU1 <jeff opcode>
TU1 <0-1 byte pad>
TU2 nIndex
```

Where the opcode translation array is:

classfile opcode	jeff opcode
wide aload	jeff_aload_w
wide astore	jeff_astore_w
wide dload	jeff_dload_w
wide dstore	jeff_dstore_w
wide fload	jeff_fload_w
wide fstore	jeff_fstore_w
wide iload	jeff_iload_w
wide istore	jeff_istore_w
wide lload	jeff_lload_w
wide lstore	jeff_lstore_w
wide ret	jeff_ret_w

4.2.11 The wide iinc Opcode

If the original instruction in class file format is:

```
TU1 wide
TU1 iinc
TU2 nIndex
TS2 nConstant
```

Where the **nIndex** value is an index to a local variable in the current frame. The **nConstant** value is a signed 16-bit constant.

The translated structure shall be the following sequence:

```
TU1 jeff_iinc_w
TU1 <0-1 byte pad>
TU2 nIndex
TS2 nConstant
```

4.2.12 Jump Opcodes

If the original instruction in class file format is:

```
TU1 <opcode>
TS2 nOffset
```

Where **<opcode>** is **goto, if_acmpeq, if_acmpne, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, ifeq, ifne, iflt, ifge, ifgt, ifle, ifnonnull, ifnull** or **jsr**. Execution proceeds at the offset **nOffset** from the address of the opcode of this instruction.

The translated structure shall be the following sequence:

```
TU1      <jeff opcode>
TU1      <0-1 byte pad>
VMOFFSET ofJump
```

Where the opcode translation array is:

classfile opcode	jeff opcode
goto	jeff_goto
if_acmpeq	jeff_if_acmpeq
if_acmpne	jeff_if_acmpne
if_icmpeq	jeff_if_icmpeq
if_icmpne	jeff_if_icmpne
if_icmplt	jeff_if_icmplt
if_icmpge	jeff_if_icmpge
if_icmpgt	jeff_if_icmpgt
if_icmple	jeff_if_icmple
ifeq	jeff_ifeq
ifne	jeff_ifne
iflt	jeff_iflt
ifge	jeff_ifge
ifgt	jeff_ifgt
ifle	jeff_ifle
ifnonnull	jeff_ifnonnull
ifnull	jeff_ifnull
jsr	jeff_jsr

The **ofJump** value is the address of the jump in the current bytecode block. It's an offset (in bytes) from the beginning of the class header structure.

4.2.13 Long Jump Opcodes

If the original instruction in class file format is:

```
TU1 <opcode>
TS4 nOffset
```

Where **<opcode>** is **goto_w** or **jsr_w.** Execution proceeds at the offset **nOffset** from the address of the opcode of this instruction.

The translated structure shall be the following sequence:

```
TU1      <jeff opcode>
TU1      <0-1 byte pad>
VMOFFSET ofJump
```

Where the opcode translation array is:

classfile opcode	jeff opcode
goto_w	jeff_goto
jsr_w	jeff_jsr

The **ofJump** value is the address of the jump in the current bytecode block. It's an offset (in bytes) from the beginning of the class header structure.

4.2.14 The sipush Opcode

If the original instruction in class file format is:

```
TU1 sipush
TU1 nByte1
TU1 nByte2
```

The translated structure shall be the following sequence:

```
TU1 jeff_sipush
TU1 <0-1 byte pad>
TS2 nValue
```

Where **nValue** is a **TS2** with the value **(nByte1 << 8) | nByte2**.

4.2.15 The newconstarray Opcode

This bytecode creates a new array with the initial values specified in the constant pool. This instruction replaces a sequence of bytecodes creating an empty array and filling it cell by cell.

```
TU1      jeff_newconstarray
VMTYPE   tArrayType
TU1      <0-1 byte pad>
TU2      nLength
VMOFFSET ofConstData
```

The **tArrayType** is a code that indicates the type of array to create. It must take one of the following values: **char[], byte[], short[], boolean[], int[], long[], float[]** or **double[]**. The **VM_TYPE_MONO** and **VM_TYPE_REF** flags are always set in this value.

The **nLength** value is the length, in elements, of the new array.

The **ofConstData** value is the offset of an array of values in the constant data section. The type of the array depends of the **tArrayType** value.

Type of Array	tArrayType Value	Structure pointed to by ofConstData
`short[]`	`0x61`	An array of **nLength JSHORT** values.
`int[]`	`0x62`	An array of **nLength JINT** values.
`long[]`	`0x63`	An array of **nLength JLONG** values.
`byte[]`	`0x64`	An array of **nLength JBYTE** values.
`char[]`	`0x65`	An Utf8 string of **nLength** characters (not prefixed by the length)
`float[]`	`0x66`	An array of **nLength JFLOAT** values.
`double[]`	`0x67`	An array of **nLength JDOUBLE** values.
`boolean[]`	`0x68`	An array of **nLength JBYTE** values. Where a zero value means **false** and a non-zero value means **true**.

A new mono-dimensional array of **nLength** elements is allocated from the garbage-collected heap. All of the elements of the new array are initialized with the values stored in the constant structure. A reference to this new array object is pushed into the operand stack.

4.3 Unchanged Instructions

This section defines all the other instruction of JEFF bytecode not previously described in section 4.2. As already noticed, these instructions are kept unchanged in the translation from class file bytecode. In order for this document to be self-contained, they are defined here.

4.3.1 One-Byte Instructions

These instructions have no argument. Here is their list (the mnemonic name of the opcode is preceded here by its value):

```
(0x00) jeff_nop
(0x01) jeff_aconst_null
(0x02) jeff_iconst_ml
(0x03) jeff_iconst_0
(0x04) jeff_iconst_1
(0x05) jeff_iconst_2
(0x06) jeff_iconst_3
(0x07) jeff_iconst_4
(0x08) jeff_iconst_5
(0x09) jeff_lconst_0
(0x0a) jeff_lconst_1
(0x0b) jeff_fconst_0
(0x0c) jeff_fconst_1
(0x0d) jeff_fconst_2
(0x0e) jeff_dconst_0
(0x0f) jeff_dconst_1
(0x1a) jeff_iload_0
(0x1b) jeff_iload_1
(0x1c) jeff_iload_2
(0x1d) jeff_iload_3
(0x1e) jeff_lload_0
(0x1f) jeff_lload_1
(0x20) jeff_lload_2
(0x21) jeff_lload_3
```

```
(0x22) jeff_fload_0
(0x23) jeff_fload_1
(0x24) jeff_fload_2
(0x25) jeff_fload_3
(0x26) jeff_dload_0
(0x27) jeff_dload_1
(0x28) jeff_dload_2
(0x29) jeff_dload_3
(0x2a) jeff_aload_0
(0x2b) jeff_aload_1
(0x2c) jeff_aload_2
(0x2d) jeff_aload_3
(0x2e) jeff_iaload
(0x2f) jeff_laload
(0x30) jeff_faload
(0x31) jeff_daload
(0x32) jeff_aaload
(0x33) jeff_baload
(0x34) jeff_caload
(0x35) jeff_saload
(0x3b) jeff_istore_0
(0x3c) jeff_istore_1
(0x3d) jeff_istore_2
(0x3e) jeff_istore_3
(0x3f) jeff_lstore_0
(0x40) jeff_lstore_1
(0x41) jeff_lstore_2
(0x42) jeff_lstore_3
(0x43) jeff_fstore_0
(0x44) jeff_fstore_1
(0x45) jeff_fstore_2
(0x46) jeff_fstore_3
(0x47) jeff_dstore_0
(0x48) jeff_dstore_1
(0x49) jeff_dstore_2
(0x4a) jeff_dstore_3
(0x4b) jeff_astore_0
(0x4c) jeff_astore_1
(0x4d) jeff_astore_2
(0x4e) jeff_astore_3
(0x4f) jeff_iastore
(0x50) jeff_lastore
(0x51) jeff_fastore
(0x52) jeff_dastore
(0x53) jeff_aastore
(0x54) jeff_bastore
(0x55) jeff_castore
(0x56) jeff_sastore
(0x57) jeff_pop
(0x58) jeff_pop2
(0x59) jeff_dup
(0x5a) jeff_dup_x1
(0x5b) jeff_dup_x2
(0x5c) jeff_dup2
(0x5d) jeff_dup2_x1
(0x5e) jeff_dup2_x2
(0x5f) jeff_swap
(0x60) jeff_iadd
(0x61) jeff_ladd
(0x62) jeff_fadd
```

```
(0x63) jeff_dadd
(0x64) jeff_isub
(0x65) jeff_lsub
(0x66) jeff_fsub
(0x67) jeff_dsub
(0x68) jeff_imul
(0x69) jeff_lmul
(0x6a) jeff_fmul
(0x6b) jeff_dmul
(0x6c) jeff_idiv
(0x6d) jeff_ldiv
(0x6e) jeff_fdiv
(0x6f) jeff_ddiv
(0x70) jeff_irem
(0x71) jeff_lrem
(0x72) jeff_frem
(0x73) jeff_drem
(0x74) jeff_ineg
(0x75) jeff_lneg
(0x76) jeff_fneg
(0x77) jeff_dneg
(0x78) jeff_ishl
(0x79) jeff_lshl
(0x7a) jeff_ishr
(0x7b) jeff_lshr
(0x7c) jeff_iushr
(0x7d) jeff_lushr
(0x7e) jeff_iand
(0x7f) jeff_land
(0x80) jeff_ior
(0x81) jeff_lor
(0x82) jeff_ixor
(0x83) jeff_lxor
(0x85) jeff_i2l
(0x86) jeff_i2f
(0x87) jeff_i2d
(0x88) jeff_l2i
(0x89) jeff_l2f
(0x8a) jeff_l2d
(0x8b) jeff_f2i
(0x8c) jeff_f2l
(0x8d) jeff_f2d
(0x8e) jeff_d2i
(0x8f) jeff_d2l
(0x90) jeff_d2f
(0x91) jeff_i2b
(0x92) jeff_i2c
(0x93) jeff_i2s
(0x94) jeff_lcmp
(0x95) jeff_fcmpl
(0x96) jeff_fcmpg
(0x97) jeff_dcmpl
(0x98) jeff_dcmpg
(0xa9) jeff_ret
(0xac) jeff_ireturn
(0xad) jeff_lreturn
(0xae) jeff_freturn
(0xaf) jeff_dreturn
(0xb0) jeff_areturn
(0xb1) jeff_return
```

```
(0xbe) jeff_arraylength
(0xbf) jeff_athrow
(0xc2) jeff_monitorenter
(0xc3) jeff_monitorexit
(0xca) jeff_breakpoint
```

4.3.2 Two-bytes Instructions

These instructions have a one byte argument. Here is their list (the mnemonic name of the opcode is preceded here by its value):

```
(0x10) jeff_bipush
(0x15) jeff_iload
(0x16) jeff_lload
(0x17) jeff_fload
(0x18) jeff_dload
(0x19) jeff_aload
(0x36) jeff_istore
(0x37) jeff_lstore
(0x38) jeff_fstore
(0x39) jeff_dstore
(0x3a) jeff_astore
```

4.4 Complete Opcode Mnemonics by Opcode

This section is the list of all the mnemonics values used in JEFF.

```
(0x00) jeff_nop
(0x01) jeff_aconst_null
(0x02) jeff_iconst_ml
(0x03) jeff_iconst_0
(0x04) jeff_iconst_1
(0x05) jeff_iconst_2
(0x06) jeff_iconst_3
(0x07) jeff_iconst_4
(0x08) jeff_iconst_5
(0x09) jeff_lconst_0
(0x0a) jeff_lconst_1
(0x0b) jeff_fconst_0
(0x0c) jeff_fconst_1
(0x0d) jeff_fconst_2
(0x0e) jeff_dconst_0
(0x0f) jeff_dconst_1
(0x10) jeff_bipush
(0x11) jeff_sipush
(0x12) jeff_unused_0x12
(0x13) jeff_unused_0x13
(0x14) jeff_unused_0x14
(0x15) jeff_iload
(0x16) jeff_lload
(0x17) jeff_fload
(0x18) jeff_dload
(0x19) jeff_aload
(0x1a) jeff_iload_0
(0x1b) jeff_iload_1
(0x1c) jeff_iload_2
(0x1d) jeff_iload_3
(0x1e) jeff_lload_0
(0x1f) jeff_lload_1
(0x20) jeff_lload_2
(0x21) jeff_lload_3
(0x22) jeff_fload_0
(0x23) jeff_fload_1
(0x24) jeff_fload_2
(0x25) jeff_fload_3
(0x26) jeff_dload_0
(0x27) jeff_dload_1
(0x28) jeff_dload_2
(0x29) jeff_dload_3
(0x2a) jeff_aload_0
(0x2b) jeff_aload_1
(0x2c) jeff_aload_2
(0x2d) jeff_aload_3
(0x2e) jeff_iaload
(0x2f) jeff_laload
(0x30) jeff_faload
(0x31) jeff_daload
(0x32) jeff_aaload
(0x33) jeff_baload
(0x34) jeff_caload
(0x35) jeff_saload
```

```
(0x36) jeff_istore
(0x37) jeff_lstore
(0x38) jeff_fstore
(0x39) jeff_dstore
(0x3a) jeff_astore
(0x3b) jeff_istore_0
(0x3c) jeff_istore_1
(0x3d) jeff_istore_2
(0x3e) jeff_istore_3
(0x3f) jeff_lstore_0
(0x40) jeff_lstore_1
(0x41) jeff_lstore_2
(0x42) jeff_lstore_3
(0x43) jeff_fstore_0
(0x44) jeff_fstore_1
(0x45) jeff_fstore_2
(0x46) jeff_fstore_3
(0x47) jeff_dstore_0
(0x48) jeff_dstore_1
(0x49) jeff_dstore_2
(0x4a) jeff_dstore_3
(0x4b) jeff_astore_0
(0x4c) jeff_astore_1
(0x4d) jeff_astore_2
(0x4e) jeff_astore_3
(0x4f) jeff_iastore
(0x50) jeff_lastore
(0x51) jeff_fastore
(0x52) jeff_dastore
(0x53) jeff_aastore
(0x54) jeff_bastore
(0x55) jeff_castore
(0x56) jeff_sastore
(0x57) jeff_pop
(0x58) jeff_pop2
(0x59) jeff_dup
(0x5a) jeff_dup_x1
(0x5b) jeff_dup_x2
(0x5c) jeff_dup2
(0x5d) jeff_dup2_x1
(0x5e) jeff_dup2_x2
(0x5f) jeff_swap
(0x60) jeff_iadd
(0x61) jeff_ladd
(0x62) jeff_fadd
(0x63) jeff_dadd
(0x64) jeff_isub
(0x65) jeff_lsub
(0x66) jeff_fsub
(0x67) jeff_dsub
(0x68) jeff_imul
(0x69) jeff_lmul
(0x6a) jeff_fmul
(0x6b) jeff_dmul
(0x6c) jeff_idiv
(0x6d) jeff_ldiv
(0x6e) jeff_fdiv
(0x6f) jeff_ddiv
(0x70) jeff_irem
(0x71) jeff_lrem
(0x72) jeff_frem
(0x73) jeff_drem
(0x74) jeff_ineg
(0x75) jeff_lneg
(0x76) jeff_fneg
(0x77) jeff_dneg
(0x78) jeff_ishl
(0x79) jeff_lshl
(0x7a) jeff_ishr
(0x7b) jeff_lshr
(0x7c) jeff_iushr
(0x7d) jeff_lushr
(0x7e) jeff_iand
(0x7f) jeff_land
(0x80) jeff_ior
(0x81) jeff_lor
(0x82) jeff_ixor
(0x83) jeff_lxor
(0x84) jeff_iinc
(0x85) jeff_i2l
(0x86) jeff_i2f
(0x87) jeff_i2d
(0x88) jeff_l2i
(0x89) jeff_l2f
(0x8a) jeff_l2d
(0x8b) jeff_f2i
(0x8c) jeff_f2l
(0x8d) jeff_f2d
(0x8e) jeff_d2i
(0x8f) jeff_d2l
(0x90) jeff_d2f
(0x91) jeff_i2b
(0x92) jeff_i2c
(0x93) jeff_i2s
(0x94) jeff_lcmp
(0x95) jeff_fcmpl
(0x96) jeff_fcmpg
(0x97) jeff_dcmpl
(0x98) jeff_dcmpg
(0x99) jeff_ifeq
(0x9a) jeff_ifne
(0x9b) jeff_iflt
(0x9c) jeff_ifge
(0x9d) jeff_ifgt
(0x9e) jeff_ifle
(0x9f) jeff_if_icmpeq
(0xa0) jeff_if_icmpne
(0xa1) jeff_if_icmplt
(0xa2) jeff_if_icmpge
(0xa3) jeff_if_icmpgt
(0xa4) jeff_if_icmple
(0xa5) jeff_if_acmpeq
(0xa6) jeff_if_acmpne
(0xa7) jeff_goto
(0xa8) jeff_jsr
(0xa9) jeff_ret
(0xaa) jeff_tableswitch
(0xab) jeff_lookupswitch
(0xac) jeff_ireturn
(0xad) jeff_lreturn
```

```
(0xae) jeff_freturn
(0xaf) jeff_dreturn
(0xb0) jeff_areturn
(0xb1) jeff_return
(0xb2) jeff_getstatic
(0xb3) jeff_putstatic
(0xb4) jeff_getfield
(0xb5) jeff_putfield
(0xb6) jeff_invokevirtual
(0xb7) jeff_invokespecial
(0xb8) jeff_invokestatic
(0xb9) jeff_invokeinterface
(0xba) jeff_unused_0xba
(0xbb) jeff_new
(0xbc) jeff_newarray
(0xbd) jeff_unused_0xbd
(0xbe) jeff_arraylength
(0xbf) jeff_athrow
(0xc0) jeff_checkcast
(0xc1) jeff_instanceof
(0xc2) jeff_monitorenter
(0xc3) jeff_monitorexit
(0xc4) jeff_unused_0xc4
(0xc5) jeff_multianewarray
(0xc6) jeff_ifnull
(0xc7) jeff_ifnonnull
(0xc8) jeff_unused_0xc8
(0xc9) jeff_unused_0xc9
(0xca) jeff_breakpoint
(0xcb) jeff_newconstarray
(0xcc) jeff_slookupswitch
(0xcd) jeff_stableswitch
(0xce) jeff_ret_w
(0xcf) jeff_iinc_w
(0xd0) jeff_sldc
(0xd1) jeff_ildc
(0xd2) jeff_lldc
(0xd3) jeff_fldc
(0xd4) jeff_dldc
(0xd5) jeff_dload_w
(0xd6) jeff_dstore_w
(0xd7) jeff_fload_w
(0xd8) jeff_fstore_w
(0xd9) jeff_iload_w
(0xda) jeff_istore_w
(0xdb) jeff_lload_w
(0xdc) jeff_lstore_w
(0xdd) jeff_aload_w
(0xde) jeff_astore_w
```

5 Restriction

The only restriction of JEFF when compared with class file format is the maximum size of a class area. Within a file, the size of a class area cannot exceed 64Kb. A class area is the block of data included between the **VMClassHeader** structure and the last data specific to the class. The JEFF syntax is very compact and the class area does not include any symbolic information. This means that the corresponding class file can be much bigger than 64Kb.

Otherwise, the following boundaries apply:

- The total size of a file cannot exceed 4Gb.
- The number of classes stored in a file cannot exceed 65,536.
- The number of packages stored in a file cannot exceed 65,536.
- The number of fields in a file cannot exceed 4Giga.
- The number of methods in a file cannot exceed 4Giga.

PAS (Publicly Available Specification) Submission

from J Consortium, Inc.

for JEFF™

11 April 2001

1. Introduction

This Explanatory Report is for J Consortium's PAS submission for JEFF. JEFF is a file format specification, which allows storing on-platform non pre-linked classes in a form that does not require any modification for efficient execution. JEFF exhibits a large range of benefits:

- The first of these benefits is that classes represented with JEFF can be executed directly from storage memory, without requiring any loading into runtime memory in order to be translated in a format adequate for execution. This results in a dramatic economy of runtime memory: programs with a size of several hundreds of kilobytes may then be executed with only a few kilobytes of dynamic runtime memory thanks to JEFF.

- The second benefit of JEFF is the saving of the processing time usually needed at the start of an execution to load into dynamic memory the stored Java™ classes.

- The third benefit is that JEFF does not require the classes to be pre-linked, hence fully preserving the flexibility of Java technology. With JEFF, programs can be updated on-platform by the mere replacement of some individual classes without requiring to replace the complete program. This provides a decisive advantage over previously proposed "ready-for-execution" formats providing only pre-linked programs.

- A last benefit of JEFF is that it allows a compact storage of programs, twice smaller than usual class file format, and this without any compression.

This submission is being made in accordance with the criteria in JTC 1 N5746, "The Transposition of Publicly Available Specifications into International Standards – A Management Guide – Revision 1 of JTC 1 N3582".

1.1 Executive Summary

JEFF is the product of J Consortium, a consortium formed in 1999, whose goal is to promote and drive the development and adoption of open, accessible Java specifications for real-time and embedded technologies. JEFF represents the synthesis of many ideas and previous product experiences in defining a storage format for representing Java classes. The goal of this new format, called JEFF, is to provide a publicly specified, royalty-free, ready-for-execution format allowing Java programs to be executed directly from static memory, thus avoiding the usual necessity to recopy classes into dynamic runtime memory for execution. Consequently, with JEFF, the size of the dynamic runtime memory needed for executing a Java program is no longer proportional to the size of the program itself and large Java programs can be executed on devices providing a dynamic runtime memory much smaller than these programs. This is especially crucial for small Java-enabled devices that cannot afford to run most applications if not using a ready-for-execution format to store Java classes. Contrary to so-called "romized" Java code, JEFF is not a pre-linked format and preserves all the flexibility of Java related to dynamic linking, which is fundamental for bringing the needed economy of memory attached to ready-for-execution format without sacrificing the flexibility of Java related to on-platform linking.

J Consortium released the JEFF specification for public review in October 2000. After including minor corrections and clarifications to the October 2000 release, the JEFF specification was finalized version 1.0, April 2001: http://www.j-consortium.org/jeffwg/jeff_spec_1.0.pdf .

J Consortium makes all of its final specifications freely available on the Internet.

1.2 Background

The usual format for storing Java classes for use on desktop and servers is called "Class File format" and is defined in published books. This is not a ready-for-execution format: the information accompanying the bytecode is stored into what is called "constant-pool". The constant-pool is viewed as a table and the references made in the bytecode to the constant-pool use indexes in this table. However, the entries of the constant-pool are of variable length and the only way to deal efficiently with this representation is to load it into dynamic memory at runtime in order to map the information into the data structures allowing an efficient exploitation. In consequence, the size of runtime memory needed to execute a Java program is always superior to the size of the stored program.

The need for a ready-for-execution format to store Java classes is obvious on small devices that have a limited amount of runtime memory. This allows an execution in-place of the code and allows to keep the runtime memory requirement very low as only the execution stack and the object heap has to reside in the runtime memory, and not the program itself.

The first kind of very small devices for which a particular idiom of Java language has ever been defined is smart cards. Smart cards are mainly used as secure payment means in many countries and are also used to secure devices such as cellular phones or set-top-boxes. In order for Java programs to fit in the memory constraints imposed by current smart card technology, a ready-for-execution format was needed and has been defined by the smart card manufacturers. This format, which is free for examination but cannot be deployed on a royalty-free basis, is called CAP File format.

The CAP File format has many interesting features and is only partially pre-linked. However, the symbolic references between classes belonging to different files are replaced by numeric tokens, which leads to reduce a part of the flexibility provided usually with Java technologies by keeping symbolic references for on-platform resolution. The CAP File format defines also numerous restrictions on the number of classes, methods and fields that can be taken into consideration for the sake of compactness of internal references.

For other kind of small devices that can support less restricted Java idioms, such as those defined for embedded systems, various so-called “romized” or “compact” format have been defined in order to provide ready-for-execution formats. However, these formats, that are generally not publicly available on a royalty-free basis, are completely pre-linked and do not provide any of the flexibility associated with the usual on-platform linking associated with Java classes.

At the end of 1999, it was clear that a ready-for-execution format was needed to help in the massive deployment of Java technologies on small devices. This format would provide all the flexibility associated with the usual Class File format and would be publicly available and deployable on a royalty-free basis.

Several small devices manufacturers had already defined, after several years of separate efforts and in a proprietary way, such ready-for-execution formats that could provide part of the flexibility provided by the usual Class File format. They decided to share their expertise in a Working Group hosted by J Consortium, contributing their IP openly to build a publicly available and royalty-free format. The best of their expertise solved the remaining issues attached to each of their proprietary formats. The final synthesis was achieved in September-October 2000 and the new format called “JEFF”.

The JEFF specification was submitted to an intensive public review and thoroughly tested, via the creation of several Class File to JEFF converters and several Java virtual machines exploiting JEFF format instead of the usual class file format. During this period of review and testing it appeared that JEFF was fulfilling all the expectations of its designers and provided even unexpected secondary benefits such as a gain of 50% on the storage size of programs with respect to usual Class File format.

It is now thought that the interest of JEFF may go beyond the world of small devices and that in fact any kind of Java virtual machine could benefit of the advantages provided by JEFF even on the biggest platforms.

1.3 Document Organization

The remainder of this Explanatory Report provides the information requested in clauses 7 of JTC1 N5746. In addition, clause 2 of this document provides information on changes to the specification during the transposition process plus a contact for questions, which may arise during any stage of the transposition process.

2. Revisions and Questions

2.1 Revisions to the Specification during Transposition Process

As JEFF is a widely implemented specification, J Consortium wants to avoid technical divergence between the transposed standard and the corresponding J Consortium specification. Therefore, in accord with JTC1 N5746, 5.2.5, the J Consortium requests that this document remain technically unchanged throughout the transposition process. If this process should uncover defects, or identify enhancements and extensions (for instance as comments accompanying ballot responses), these will be collected and processed as stated in 3.1 below.

2.2 Questions During the Voting Period

J Consortium welcomes any questions NBs may have during the voting period for this PAS submission and will endeavor to reply promptly. Questions should be addressed to:

Wendy Fong
Chairman, J Consortium
P.O. Box 1565
Cupertino, California 95015-1565.

or by e-mail to

wendy_fong@hp.com

3. Organization (J Consortium's) Acceptance Criteria

As requested in N5746, 5.2.2, the conditions for J Consortium's recognition as a PAS submitter have not changed. This information is documented in J Consortium's PAS submitter application, document JTC1 N6088, "J Consortium's Application to ISO/IEC JTC 1 for recognition as a Submitter of a Publicly Available Specification (PAS) (available on the JTC1 web site at http://www.jtc1.org). This application was approved as documented in JTC1 N6316, "Notice of Approval of the Application from the J Consortium for Recognition as a Submitter of Publicly Available Specifications" following the response to the ballot comments submitted to JTC 1 in "J Consortium's Response to ISO/IEC JTC 1 N 6178 for recognition as a Submitter of a Publicly Available Specification (PAS)".

The material in these documents is not duplicated in this clause.

3.1 Comments Addressed from PAS Application

Many of the original ballot comments concerned the Document Related Criteria which are addressed below in clause 4. Of course, the J Consortium's "track record" is now one year more mature than when the PAS application was originally delivered. And J Consortium continues to have a cooperative attitude about any Working Arrangements that may be proposed by JTC 1. Also, there are no third-party trademarks in the title of this specification.

Enhancements, extensions, and major modifications to J Consortium specifications are handled procedurally in J Consortium in much the same way they are in ISO, i.e., new work. Any related new J Consortium specifications will be promptly submitted to JTC 1 using the PAS process to progress a new ISO standard. Again, this will maintain alignment of the ISO standard and the J Consortium specification.

3.2 J Consortium's By-laws, SD-2 Technical Committee Organization, Rules, and Procedures and Membership Information

There have been no updates to J Consortium's By-laws or SD-2 Technical Committee Organization, Rules, and Procedures documents since the 2000 application in JTC 1 N6088 and the J Consortium Membership has grown. These documents are available for viewing at: http://www.j-consortium.org

4. Document Related Criteria

The criteria requested in 7.4 of JTC 1 N5746 are addressed in the following subclauses. The numbering is identical to that in 7.4 without the leading 7 and the items of information requested are duplicated below with the items of information being requested rendered in italics.

4.1 Quality

Within its scope the specification shall completely describe the functionality (in terms of interfaces, protocols, formats, etc) necessary for an implementation of the PAS. If it is based on a product, it shall include all the functionality necessary to achieve the stated level of compatibility or interoperability in a product <u>independent</u> manner.

4.1.1) Completeness (M):

a) How well are all interfaces specified?

JEFF is a format that is clearly and completely specified in the specification document provided by J Consortium. The specification document is self-contained and can be read as a grammar specification. In addition, information is provided that allows creation of a Class File to JEFF converter.

b) How easily can implementation take place without need of additional descriptions?

JEFF specification defines completely a format in a self-contained way. With no additional description than those provided in the specification, it is possible to implement a JEFF analyzer that can test the compliance of any file with the specified format.

The way information is represented in JEFF and how to navigate in a JEFF file to retrieve information is completely defined in a self-contained manner.

The specification provides in addition sufficient information to create a converter translating sets of Class Files into JEFF files.

c) *What proof exists for successful implementations (e.g. availability of test results for media standards)?*

At present, there are at least three successful commercial tools from different vendors that will be available in the market that support JEFF. These tools include a Class File to JEFF converter and a virtual machine using exclusively JEFF for program representation. Two of these virtual machines have already passed a very exhaustive suite of more than 500 test programs made for virtual machines using the usual Class File format.

4.1.2) Clarity:

a) *What means are used to provide definitive descriptions beyond straight text?*

The specification is a semi-formalized text with numerous tables that defines completely the grammar of the format.

b) *What tables, figures, and reference materials are used to remove ambiguity?*

The specification uses numerous tables in addition to semi-formalized textual definitions. The specification is made in such a way that no syntactic ambiguity can remain. In addition, the specification provides all the information needed to create a Class File to JEFF converter, which fixes any semantic ambiguity for the reader aware of the domain.

c) *What contextual material is provided to educate the reader?*

An additional slide presentation is provided on the J Consortium web site, written by some of the main designers of JEFF:
http://www.j-consortium.org/jeffwg/jeffpres0201.pdf

On the public web site of one of the vendors of JEFF-related tools can be found a white paper explaining the advantages of JEFF:
http://www.cardsoft.com/html/products/CJKVM_WP_v1.11.PDF

and a slide presentation of various products using JEFF technology:
http://www.cardsoft.com/html/events/presentations/RoadshowSlidesJPB.pdf.

4.1.3) Testability (M)

The extent and use of conformance/interoperability tests or means of implementation verification (e.g. availability of reference material for magnetic media) shall be described, as well as the provisions the specification has for testability.
The specification shall have had sufficient review over an extended time period to characterize it as being stable.

Currently, there are no formal JEFF testing organizations such as The Open Group or NIST. However, throughout the specification, is clearly and completely specified what is necessary for an implementation to claim compliance or partial compliance to the specification.

The initial draft was available for 6 months and has received extensive scrutiny and feedback from a large user community.

Also, as noted earlier, there are over three commercial realizations of the standard, one of them made by a company that was not involved in the design of JEFF, and some academic and research realizations, which serve as confirmation of the validity and feasibility of JEFF.

4.1.4) Stability (M):

a) How long has the specification existed, unchanged, since some form of verification (e.g., prototype testing, paper analysis, full interoperability tests) has been achieved?
b) To what extent and for how long have products been implemented using the specification?
c) What mechanisms are in place to track versions, fixes, and addendums?

The initial specification was available for six months and has received extensive scrutiny and feedback from a large user community.

As noted earlier, there are over three commercial realizations of the standard and, in addition, many academic and research realizations, which serve as confirmation of the validity and feasibility of JEFF.

Fixes and new versions are handled in accord with the J Consortium process described briefly in 3.1 above and full detail in the J Consortium SD-2 Technical Committee Organization, Rules and Procedures, see 3.2.

4.1.5) Availability (M):
a) Where is the specification available(e.g., one source, multinational locations,

what types of distributors)?

This specification is available on the Internet from the J Consortium web site at http://www.j-consoritum.org/jeffwg/index.html.

b) How long has the specification been available?

The JEFF specification has been available since April 2001.

c) Has the distribution been widespread or restricted? (describe situation)
d) What are the costs associated with specification availability?

This specification is distributed with no restrictions and at no cost from the J Consortium web site.

4.2 Consensus (M)

The accompanying report shall describe the extent of (inter)national consensus that the document has already achieved.

4.2.1) Development Consensus:

a) Describe the process by which the specification was developed.
b) Describe the process by which the specification was approved.
c) What "levels" of approval have been obtained?

This, and all J Consortium specifications, are created under the open process set forth in the J Consortium SD-2 Technical Committee Organization, Rules and Procedures, see reference in 3.2. This specification was approved by J Consortium members from all major industrialized countries and many developing countries. J Consortium membership is open to all organizations.

4.2.2) Response to User Requirements:

a) How and when were user requirements considered and utilized?
b) To what extent have users demonstrated satisfaction?

Users are involved in the process throughout except for the portion of the process requiring commercialization in products. During the review process of the JEFF specification several implementations of Class File to JEFF converters and JEFF-based virtual machines have been completed and have been reported to give entire satisfaction to their implementers after small modifications were made to the specs to correct minor inconsistencies detected during implementation.

4.2.3) Market Acceptance:

a) How widespread is the market acceptance today? Anticipated?
b) What evidence is there of market acceptance in the literature?

JEFF is a new format that provides incontestable and quite needed benefits with no drawback. Being defined independently, JEFF has not yet received a massive promotion. However, JEFF is already supported by independent non-profit organizations that are publicly providing APIs for small devices, such as the STIP Consortium (www.stip.org) or FINREAD, a workshop of CEN/ISS (final specifications to be published on May 15).

4.2.4) Credibility:

a) What is the extent and use of conformance tests or means of implementation verification?

Conformance tests can be fully implemented. In practice, existing converters provide automatically tested JEFF files as a result of the strict application of a translation conforming to the grammar defined in the specification.

b) What provisions does the specification have for testability?

Full testability is granted by the specification as it defines completely the format. In practice, JEFF files are coming from a translation from a set of Class files resulting from the application of a converter applying the grammar defined in the specification. Such files resulting from a conversion are necessarily fully compliant with the specification if the converter has been developed according to the specification, which is the case of the existing commercial converters.

4.3 Alignment

The specification should be aligned with existing JTC1 standards or ongoing work and thus complement existing standards, architectures and style guides. Any conflicts with existing standards, architectures and style guides should be made clear and justified.

4.3.1) Relationship to Existing Standards:
a) What international standards are closely related to the specification and how?

There are none known.

b) To what international standards is the proposed specification a natural extension?

There are none known.

c) How is the specification related to emerging and ongoing JTC 1 projects?

There are none known.

4.3.2) Adaptability and Migration:

a) What adaptations (migrations) of either the specification or international standards would improve the relationship between the specification and international standards?
b) How much flexibility do the proponents of the specification have?

The specification can be readily adapted to a number of other standards through the use of its specialization (extensibility) mechanisms. With these mechanisms, it is possible to create refined versions of the general JEFF specifications for specific purposes or specific domains.

c) What are the longer range plans for new/evolving specifications?

J Consortium plans to submit new/evolving specifications to JTC 1.

4.3.3) Substitution and Replacement:

a) What needs exist, if any, to replace an existing international standard? Rationale?

There are none known.

b) What is the need and feasibility of using only a portion of the specification as an international standard?

There are none known.

a) What portions, if any, of the specification do not belong in an international standard (e.g., too implementation specific)?

There are none known.

4.3.4) Document Format and Style

a) What plans, if any, exist to conform to JTC 1 document styles?

The document is in J Consortium's document format, but we are unaware of any substantive incompatibilities with ISO's document format.