Specifying trace formats: MetaTF 1.2.1

Richard Jones
Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF
R.E.Jones@ukc.ac.uk

March 30, 2001

1 Background

This document represents the current definition of Meta-TF, version 1.2. It has been developed from Meta-TF 1.0 after discussion at ISMM 2000 in Minneapolis, October 15-16, 2000. (Meta-TF 1.1 was a simplification of Meta-TF 1.0, designed to simplify tool construction).

1.1 Changes from Meta-TF 1.0

The original definition of Meta-TF 1.0 can be found in *Designing a Trace Format for Heap Allocation Events*, Trishul Chilimbi, Richard Jones and Benjamin Zorn, ISMM 2000, Minneapolis, October 2000, ACM.
Following discussion at ISMM 2000, Meta-TF 1.2 makes the following changes:

1. The trace format is ASCII, one line per record. Although binary encodings may offer greater compression, the difference is slight and does not outweigh the benefit of being able to debug ASCII traces or to use them with other text-based tools.

   In the future, Meta-TF may also offer an XML format. This option is likely to become more attractive as more parsers and compressors for XML appear. In particular, we have been impressed by the compression offered by AT&T's xmill tool (which uses split-stream encoding on XML elements).

2. Meta-TF 1.2 offers *cached* sections as well as plain sections. The intention of cached sections is that they may be used to look up attributes (such as types or call-stacks) of records in other sections. Meta-TF reader tools are expected to store cached records and make caches available to client programs. Sections may be interleaved. Other than the optional keyword, `cache`, the syntax for cached and other sections is identical; both share the same name- and number-space.

3. Record fields are typed. The primitive types are `byte`, `short`, `int`, `long`, `string` or `index`. All values in a *trace* are either hexadecimal integers or strings. Integer literals in a *dtd* may be either decimal or hecadecimal; the latter are written as `0x....` `tag` fields have an implicit type `int`.

4. The record field property 'interpretation' is renamed 'encoding'.

5. Record fields no longer have widths (since the format is ASCII). The effect of setting a field's width to 0 may be achieved by setting its encoding to one with an implicitly zero width, i.e. `Default` (recommended) or `Stride`. Strings may only have encodings `None` or `Default`. A field of type `index` must have encoding `sectionOffset`, `sectionStride` or `default 0`.

6. The syntax for defining fields is changed. For example, a field named *foo*, which has a default integer value of 4, is defined by

   ```
   int foo = default 3;
   ```

7. Trace files are expected to contain conventional comments giving the provenance of the trace.

8. The built-in record `vfield` is no longer provided.

9. Metadata records may change either a given field of a single record type in the current section, or they may change all fields of a given name in all records of the current section.

10. DTD names are case-*in*sensitive in order to allow the code generators greater freedom in choice of capitalisation of class and member names.

1.2 Changes from Meta-TF 1.1

Meta-TF 1.1 was a simplification of Meta-TF 1.0, designed to allow more efficient allocators to be generated, and to clear up ambiguities in the original specification. MetaTF 1.1 was used to build the tools reported on at ISMM 2000. Changes in Meta-TF 1.1

1. Removed RHS of property in the form `field.property`. Consequently, `value` is no longer a reserved name. `vfield` is now defined using the "*" syntax.

2. Removed anonymous tuples, e.g. `rec : (a, b, (x, y), c)`. These must now be written as either

   ```
   event : (a, b, x, y, z)
   ```

   or

   ```
   xy : (x, y)
   event: (a, b, xy, c)
   ```

3. In repeated fields, such as `a*b`, `a` is a new field, not a reference to an existing one (this was unclear in Meta-TF 1.0).

4. The width of the tag field is fixed at one signed byte. Negative values and zero are reserved for built-in values.

2 The Trace File

A trace file consists of a header, followed by a list of records defined by the DTD. The header is expected to comprise a list of required and optional conventional comments that describe the trace and its provenance.

3 The DTD

3.1 Sections

The DTD defines the sections and record types that may appear within a valid trace data file.

```
<docTypeDef> ::= <section0> <section>*
```

Comments in the both the DTD and the trace follow perl / Unix shell style of a single line comment introduced by $\sharp$.

```
<section0> ::= <defs>?
<section> ::= "cached"? "section"  <sectionName>  <sectionNumber>
                   "{" <defs> "}"
<defs> ::= { <record> | <property> | <comment> }*
```

The intention of cached sections is that they may be used to look up attributes of records (such as types or call-stacks) in other sections. Meta-TF reader tools are expected to store cached records and make these available to client programs.

3.2 Records

The DTD defines, for each record, the fields that that record contains.

```
<record> ::= <recordName> ":" <field>
                 [ "{" { <type>? <property>* }* "}" ]
```

3.3 Fields

The field name `tag` is reserved. Its encoding (see below) is always `none`, and a *value* (see below) must be provided. The value must be unique in this section and must not conflict with values of built-in records (see below).

The scope of an record definition is global if defined in section 0, or limited to the named section within which the record is defined, otherwise. Other than the implied global section 0, `Global`, each section shall have a unique *name* and a unique, positive *ID*. It is expected that section IDs will be small natural numbers. `section` and `cached` are reserved names. All names are statically scoped and are case-*in*sensitive.

3.3.1 Types

```
<type> ::= "byte" | "short" | "int" | "long" | "string" | "index"
```

All user-defined fields must be assigned a type (i.e. the only field for which the type may be omitted is `tag`, whose implicit type for user-defined records is `int`). Types of event fields may not be changed by metadata. Primitive fields in the trace contain either hexadecimal integers or strings. By giving an integral type for a field, the DTD guarantees that trace elements of this type may be stored in a programming language value of corresponding type, i.e. byte (1 byte), short (2 bytes), int (4 bytes) and long (8 bytes). Strings are defined in the usual way (delimited by "s, may not contain new lines, the usual C/C++/Java escape codes, \n, \\, etc.) `index` is used to denote that the non-negative integer value of this field is an index into a cached section. It is a run-time error if this value exceeds the number of records stored in the section at the point this field is encountered in a trace. A field of type `index` must have encoding `sectionOffset`, `sectionStride` or `default 0`.

```
<field> ::= <tupleField> | <simpleField>
<simpleField> :: = <fieldName> | <repeatField>
<repeatField> ::= <fieldName> "*" <fieldName>
<tupleField> :: = "(" <field> { "," <field> }* ")"
```

The `fieldName*fieldName` syntax is used to handle a sequence of fields. The value found in the `fieldName` field of each event indicates the number of occurrences of `namedField` that follow. The `(field,field,...,field)` is used to denote composite fields.

3.3.2 Properties

```
<property> ::= <fieldName> { "." <propertyName> }? "=" <value> ";"
```

Properties may be defined for each field of an event. Property names are local to each event definition. Use of a field places an obligation to define, for that field, a *type* (except for built-in fields) and an *encoding* property (unless these are already defined for this section or globally). Tag fields may only appear as the first field in a record definition; they must have a `value` property. Standard property names are: `value | encoding`.

Two default properties are provided (allowing the `"." <propertyName>` to be omitted). For tag fields, the default is `value`; for non-tag fields, the default is `encoding`. Thus an event may be defined by:

```
free : (tag, address) { tag = 7; int address = none; }
```

Field encodings follow those of HATF 1.0. Encodings of fields with encodings `default`, `stride` and `sectionStride` use zero bytes in the trace.

```
<value> ::= "none" | "baseOffset" <base> | "delta" <initialValue> |
            "default" <defaultValue> | "stride" <initialValue> <stride> |
            "sectionOffset" <sectionName> | "sectionStride" <sectionName> <stride>
```

`value` parameters other than `sectionName` are decimal or hexadecimal integers. The latter are written in the form `0x...`. All numerical values in *traces*, however, are represented by hexadecimal integers.

3.4 Built-in records

3.4.1 Comments

Three records are predefined; their names are reserved. `comment` is provided as a built-in simply because it is ubiquitous.

```
comment : (tag, text) { tag = '♯'; string text = none; }
```

Trace files are expected to contain conventional comments giving the provenance of the trace. The first character of a conventional comment after its tag is a ♯. The following conventional comments are required:

```
♯♯ TraceName: the name of the trace
♯♯ Author: the author of the trace
♯♯ Date: the date (UTC) the trace was generated
♯♯ DTDURL: the URL of the DTD defining the trace
♯♯ Description:
```

URL gives the URL of the DTD defining the trace. The following conventional comments are optional:

```
♯♯ Program: the program executed
♯♯ ProgramURL: the location of program executed
♯♯ Arguments: command line arguments
♯♯ CompilerFlags: the flags used to compile the program
♯♯ OS: the operating system and version
♯♯ Hardware: hardware configuration, e.g.  processor, cache, memory, etc.
♯♯ TraceURL: the URL of the trace itself
```

3.4.2 Section records

`section` indicates that the trace has switched section.

```
section : (tag, number) { tag = '$'; int number = none; }
```

3.4.3 Metadata

`metadata` records may be used to change the encoding of user-defined records on the fly. Note that index types cannot have their encoding changed by metadata if the DTD specifies their encoding to be `default`: the section to which index types refer is fixed at compile-time.

```
metadata : (tag, record, field, encoding, value...) {
  tag = '!';
  record = none; ♯ tag of record to be changed, * if all records in current section
  field = none; ♯ number of the record to be changed (0..)
  encoding = none; ♯ new encoding for this field
                       ♯ number of value fields depends on the propertyName
  ♯------------------------------------------------------------------♯
  ♯  encoding effect                       value...            ♯
  ♯------------------------------------------------------------------♯
  ♯  0              encoding=baseOffset    base                ♯
  ♯  1              encoding=default       value               ♯
  ♯  2              encoding=delta         initialValue        ♯
  ♯  3              encoding=none          --                  ♯
  ♯  4              encoding=sectionOffset --                  ♯
  ♯  5              encoding=sectionStride initialValue, stride ♯
  ♯  6              encoding=Stride        initialValue, stride ♯
  ♯------------------------------------------------------------------♯
}
```

Either the encoding of a *specific* field of a *specific* record may be changed, or the encoding of *every* instance of a named field in *every* record in the *current* section may be changed.

`record=*`: All fields corresponding to `field` are to be changed.

`record=`*n*: `field` gives the number of the field in record, with tag n, to be changed (field numbers start with the tag, whose field number is 0).

In the non-`*` tag case ("set a field for this event type"), each field is associated with its lexical position in a record, i.e. the event definition is flattened. We believe that this is implementationally simple and fairly intuitive, although it requires some care by the user (they must count their way through records and sub-records). The MetaTFtool, for example, generates symbolic names for these field positions (such as `TAG`).

In the `*` case ("set all fields"), fields are enumerated by their appearance in the DTD, starting at 0. The MetaTFtool constructs such an enumeration as it reads the DTD, using the order in which it encounters fields, and provides symbolic names for these encodings (such as `ALL_TAG`).

The number of arguments of a `metadata` record is encoded in the `encoding` field. If the encoding set is `None` or `sectionOffset`, then no value is given; if the encoding is `Stride` or `sectionStride`, then both an initial value and a stride is given; otherwise, a single value is given. For this reason and because of the values found in the `record` field, it is not possible to express `metadata` records in Meta-TF syntax (unlike other built-in event types).

4 Tutorial

This simple tutorial provides a definition of a simple DTD, an explanation of how to use MetaTFtool, a compiler for MetaTF that translates DTD definitions into a Java trace reader/writer, and a simple client that uses this reader.

4.1 A simple DTD

HATF 1.0 can be defined with Meta-TF 1.2.

```
♯ HATF10.dtd: A DTD for HATF 1.0
♯ Meta-TF version 1.2

int size    = none;
int address = none;
string attributes = default "";
int time   = default 0;
int thread = default 0;
int heap   = default 0;

section heap 1 {
  alloc :            (tag, size, address, time, thread, heap, attributes)          { tag = 1; }
  free :             (tag, address, time, thread, heap, attributes)                { tag = 2; }
  reallocNoAlloc :   (tag, size, address, address, time, thread, heap, attributes) { tag = 3; }
  reallocAllocFree : (tag, size, address, address, time, thread, heap, attributes) { tag = 4; }
  reallocAlloc :     (tag, size, address, address, time, thread, heap, attributes) { tag = 5; }
  reallocFree :      (tag, size, address, address, time, thread, heap, attributes) { tag = 6; }
  createHeap :       (tag, thread, heap, attributes)                               { tag = 7; }
  destroyHeap :      (tag, thread, heap, attributes)                               { tag = 8; }
  createThread :     (tag, thread, attributes)                                     { tag = 9; }
  destroyThread :    (tag, thread, attributes)                                     { tag = 10; }
}
```

4.2 MetaTFtool

MetaTFtool is a tool for generating a reader/writer and classes representing each section and record of a DTD. Its synopsis is

```
java MetaTFtool -J directory < dtd
java MetaTFtool -J directory dtd
```

directory is the name of a directory in which the generated source code for each class will be placed. *dtd* is the name of the DTD. Note that `MetaTF` must be in the classpath. If the HATF DTD is in `hatf.dtd`, and we say

```
java MetaTFtool -J Hatf hatf.dtd
```

MetaTF will generate the following class structure:

```
Hatf/
  Hatf.java                    the reader/writer
  Global.java                  section 0
  Heap.java                    section 1
  Heap_Alloc.java              the records...
  Heap_Free.java
  Heap_ReallocNoAlloc.java
  Heap_ReallocAllocFree.java
  Heap_ReallocAlloc.java
  Heap_ReallocFree.java
```

```
Heap_CreateHeap.java
Heap_DestroyHeap.java
Heap_CreateThread.java
Heap_DestroyThread.java
```

where `Hatf.java` is the reader/writer class, `Global.java` represents the global section 0, `Heap.java` represents the section 1 and `Heap_*.java` are the classes representing each record in section heap.

4.2.1 A simple HATF reader

A simple reader for a trace defined by the HATF DTD might be:

```
import Hatf.*;
import MetaTF.Runtime.*;
import MetaTF.BadRecordException;
import MetaTF.MetaTFConstants;
import java.io.*;
class TestDriver {
  public static void main(String[] args) {
    try {
      LowLevelReader rdr = new AsciiReader(new FileInputStream("test.trace"));
      Hatf.init();
      int i = 0;
      do {
        try {
          Record r = Test.getNextRecord(rdr);
          //...process r...
          i++;
        } catch (BadRecordException e) {
          System.err.println(e.getMessage()+" at record "+i);
        } catch (IOException e) {
          System.err.println(i+" records read.");
          System.exit(0);
        }
      } while(true);
    } catch(BadRecordException e) {
      System.err.println(e.getMessage());
    } catch(IOException e) {
      System.err.println(e.getMessage());
    }
  }
}
```

5 The Java API

The MetaTF tool generates trace readers and writers from a DTD. Currently the tool can generate these in the following languages: Java. This section documents the API for the classes provided by MetaTFtool.

5.1 Interpretation constants

The encoding may be any of

```
MetaTF.MetaTFConstants.BaseOffset
MetaTF.MetaTFConstants.Default
MetaTF.MetaTFConstants.Delta
MetaTF.MetaTFConstants.None
MetaTF.MetaTFConstants.SectionOffset
MetaTF.MetaTFConstants.SectionStride
MetaTF.MetaTFConstants.Stride
```

5.2 Records

Each record defined in the DTD is represented by a subclass of the abstract class `MetaTF.Runtime.Record`. Thus, if the DTD specifies a record *alloc* in the section *heap*, a subclass `Heap_Alloc` (note capitalisation to distinguish class names from instance names) will be generated. `Heap_Alloc` will contain public attributes whose names are those of the fields of *alloc*. If a field has type `index`, the associated attribute of the class will be a reference to a record in the appropriate cached section.

5.2.1 Record constructors

Two constructors are generated for each record, one that takes no arguments and one that takes an argument for each field of the record.

5.2.2 Writing records

MetaTFtool generates the method

```
public abstract void putRecord(LowLevelWriter llw) throws IOException;
```

that writes a record through a `LowLevelWriter`, encoding its fields appropriately.

5.2.3 Per-record metadata

Methods to write metadata records to change the encoding of fields of a record are also generated:

```
public static void putMetadata(LowLevelWriter llw, int f, int i, long v)
       throws BadRecordException, IOException
public static void putMetadata(LowLevelWriter llw, int f, int i, long v1, long v2)
       throws BadRecordException, IOException
public static void putMetadata(LowLevelWriter llw, int f, int i, String v)
       throws BadRecordException, IOException
```

Writing a metadata record has the side-effect of changing the reader/writer's state appropriately. For example,

```
Alloc.putMetata(lw, MyRecord.TIME, MetaTFConstants.Default, 0);
```

both writes a metadata record to change the encoding of the *time* field of record *alloc* to *default 0* and updates the writer's internal state so that future *alloc* records omit the *time* field. Fields are numbered by their order in the DTD definition of records, after any expansion of sub-records has taken place. For each field, a constant representing this encoding is generated. Field names always start with a lower-case letter; repeated names are disambiguated by appending an integer (starting from 2). The constant identifying a field is represented by the all upper-case version of the field name. For example, for the field `tag`, the constant would be `TAG`.

5.2.4 `Comment`

One built-in record is provided: `MetaTF.Runtime.Comment`:

```
public class Comment extends Record {
  public static final int  sectionNumber = 0;
  public static final int  TAG = 0;
  public static final char tag = MetaTFConstants.Comment;
  public static final int  TEXT = 1;
  public String            text;

  public Comment() {}
  public Comment(String text);

  public void putRecord(LowLevelWriter llw) throws IOException;
}
```

5.3 Sections

Each section in the DTD is represented by a subclass of the abstract class `MetaTF.Runtime.Section`.

5.3.1 Section numbers

Each section has a number.

```
public static final int sectionNumber
```

5.3.2 Cached sections

Sections may be cached.

```
public final boolean iscached
```

If and only if a section is cached, methods are provided to add a record to its cache, to extract a record from its cache and to retrieve the whole cache.

```
public static void cache(Record r)
public static Record getRecord(int i) throws BadRecordException
public static List getCache()
```

5.3.3 Per-section metadata

Each section provides methods to write metadata records that change the encoding of *all* fields of a given name within the section.

```
public static void putMetadata(LowLevelWriter llw, int f, int i)
       throws BadRecordException, IOException
public static void putMetadata(LowLevelWriter llw, int f, int i, long v)
       throws BadRecordException, IOException
public static void putMetadata(LowLevelWriter llw, int f, int i, long v1, long v2)
       throws BadRecordException, IOException
public static void putMetadata(LowLevelWriter llw, int f, int i, String v)
       throws BadRecordException, IOException
```

For example, to change the encoding of all *time* field within section *heap* to *none*, emit the metadata record:

```
Heap.putMetadata(llw, Hatf.ALL_TIME, MetaTFConstants.None);
```

Note that the 'all fields' encodings are defined globally, rather than per section.

5.4 The reader/writer class

MetaTFtool generates a reader/writer for each DTD. The reader/writer provides an initialisation method `init` to initialise the reader/writer, a `getNextRecord` method that returns the next non-special record in the trace, a `putSection` method to write a record changing sections (after calling its `setSection` method to change the reader/writer's internal state) and an encoding of all field names in the DTD.

5.4.1 Initialisation

The first action of a client of a reader or writer should be to initialise it by calling its `init` method. `init` may be called again to re-initialise the reader/writer, for example to read or write a further trace.

5.4.2 Reading records

The main interface is the method `getNextRecord` that returns the next non-special record in the trace. Section and metadata records are special: they affect the state of the reader. If a special record is encountered, the reader adjusts its state and reads the next record (which may in turn be special). The reader will store all records of a cached section in the appropriate section object.

5.4.3 Per-section field encodings

Because `MetaTF.Runtime.Section` needs to be able to change the encoding of every instance of a named field within a section, the reader/writer provides an encoding of each field in the DTD. These field names are encoded as fields by capitalising the field name and adding the prefix `ALL_`. Thus *all* fields named `tag` within a section are encoded as `ALL_TAG`.

5.4.4 Switching sections

`putSection` writes a record changing sections (after calling its `setSection` method to change the reader/writer's internal state).

5.5 Exceptions

Methods may throw `java.io.IOException` or `MetaTF.BadRecordException` if an error occurs. Instances of `java.io.IOException` are thrown if an error is encountered trying to read or write a file; `MetaTF.BadRecordException` is thrown if an attempt is made to change an encoding to an invalid value, or if an incorrectly formed record is encountered.

5.6 Internal classes and methods

Some classes and methods are largely internal to the reader/writer. They are not intended to be called directly by client programs, but are used by methods generated by MetaTFtool. They are documented here for completeness.

5.6.1 Records

All records are inherited from the abstract class `MetaTF.Runtime.Record`. The methods provided by this class are never normally invoked directly. As well as the abstract method `putRecord`, `MetaTF.Runtime.Record` also provides methods to write record fields according to their encoding: None of these methods should normally be called directly by the user.

```
protected void write(LowLevelWriter llw, NumTraceField tf, long v)
       throws IOException
protected void write(LowLevelWriter llw, IndexTraceField tf, int v)
       throws IOException
protected void write(LowLevelWriter llw, StringTraceField tf, String v)
       throws IOException
```

It also provides static methods for writing metadata (and adjusting the internal state of the writer accordingly); these methods are called by the `putMetadata` methods of the concrete classes (which provide the section number and record tag parameters):

```
protected static void putMetadata(LowLevelWriter llw, int s, int t, int f, int i)
          throws BadRecordException, IOException
protected static void putMetadata(LowLevelWriter llw, int s, int t, int f, int i, long v)
          throws BadRecordException, IOException
protected static void putMetadata(LowLevelWriter llw, int s, int t, int f, int i, long v1, long v2)
          throws BadRecordException, IOException
protected static void putMetadata(LowLevelWriter llw, int s, int t, int f, int i, String v)
          throws BadRecordException, IOException
```

5.6.2 Sections

All sections are inherited from the abstract class `MetaTF.Runtime.Section`. This class provides methods to initialise the reader/writer, to change the encoding of all fields in a section and to write the appropriate metadata record,

The encoding of all instances of a named field within a section may be changed in a similar manner to the way in which a record's encoding may be changed. Fields are numbered by the order of their appearance in the DTD. None of these methods should normally be called directly by the user.

```
static void changeInterpretation(int secno, int field, int enc)
       throws BadRecordException
static void changeInterpretation(int secno, int field, int enc, long v1)
       throws BadRecordException
static void changeInterpretation(int secno, int field, int enc, String v)
       throws BadRecordException
static void changeInterpretation(int secno, int field, int enc, long v1, long v2)
       throws BadRecordException

protected static void putSectionMetadata(LowLevelWriter llw, int s, int f, int i)
       throws BadRecordException, IOException
protected static void putSectionMetadata(LowLevelWriter llw, int s, int f, int i, long v)
       throws BadRecordException, IOException
protected static void putSectionMetadata(LowLevelWriter llw, int s, int f, int i, long v1, long v2)
       throws BadRecordException, IOException
protected static void putSectionMetadata(LowLevelWriter llw, int s, int f, int i, String v)
       throws BadRecordException, IOException
```

The code generator also provides code to initialise readers/writers: None of these methods should normally be called directly by the user.

```
static void initLookup(int n, int f)
public static void initLookupList(int s, int n)
static void addLookupField(int s, int f, TraceField tf)
```

5.6.3 Low-level readers and writers

Most methods that perform IO do so through classes that implement the interfaces `LowLevelReader` and `LowLevelWriter`. The methods in these interfaces do not understand how to decode fields —

that function is carried out by higher level methods such as `MetaTF.Runtime.Record.write` — but simply read or write tokens. The interfaces are:

```
public interface LowLevelReader {
    public long   readNum()    throws BadRecordException, IOException;
    public int    readTag()    throws BadRecordException, IOException;
    public String readString() throws BadRecordException, IOException;
    public String readAny()    throws BadRecordException, IOException;
    public String readln()     throws BadRecordException, IOException;
}

public interface LowLevelWriter {
    public void write(long value)   throws IOException;
    public void write(int value)    throws IOException;
    public void write(String value) throws IOException;
    public void write(char value)   throws IOException;
    public void writeln()           throws IOException;
    public void writeln(String s)   throws IOException;
    public void flush();
    public void close();
}
```

A few are worth commenting on. `readAny` is used to read a special character without encoding; it is only used internally to read the record field (which may be an integer or `*`) of a metadata record. `readln` is used to read the text of a comment, up to and including the new line. `writeln()` terminates a record; `writeln(String s)` is used to write the text of a comment.

5.7 A test program

This program simply writes a trace and then reads it back. (It needs test data to exercise the `sectionStride` encoding.)

```
# Test.dtd for MetaTF 1.2

cached section S1 1 {
  one : (tag, desc) { tag = 1; string desc = none; }
}

section S2 2 {
  two : (tag, a, a, b, cmt) { tag=2; int a = none; long b = none; string cmt = none; }
  three : (tag, a, b, cmt) { tag=3; int a = none; index b = sectionOffset S1; string cmt = none; }
}
```

A test driver...

```
import Test.*;
import MetaTF.Runtime.*;
import MetaTF.BadRecordException;
import MetaTF.MetaTFConstants;
import java.io.*;

class TestDriver {
  public static void main(String[] args) {
    try {
      LowLevelWriter wtr =
          new AsciiWriter(new PrintStream(new FileOutputStream("test.trace")));
      Test.init();
      new Comment("# TraceName: TEST TRACE for MetaTF 1.2").putRecord(wtr);
      new Comment("# Author: Richard Jones (R.E.Jones@ukc.ac.uk)").putRecord(wtr);
      new Comment("# Date: Thu Dec 12 11:12:28 GMT 2000").putRecord(wtr);
      new Comment("# DTDURL: MetaTF/dtd/test.dtd").putRecord(wtr);
      new Comment("# Description: Format - each record includes contains <record>.putRecord(wtr); "+
                  "trace: <expected trace record>").putRecord(wtr);

      new Comment("Test.putSection(wtr,1); trace: $ 1").putRecord(wtr);
      Test.putSection(wtr,1);

      // Set up section one
      new Comment("Some section data, 20 records").putRecord(wtr);
      S1_One[] ones = new S1_One[20];
      for(int i=0; i<20; i++) {
        ones[i] = new S1_One("S1_One(cmt).putRecord(wtr); trace: 1 ");
        ones[i].putRecord(wtr);
      }

      new Comment("Test.putSection(wtr,2); trace: $ 2").putRecord(wtr);
      Test.putSection(wtr, 2);

      new S2_Two(1, 10, 100, "S2_Two(1,10,100,cmt).putRecord(wtr); "+
                             "trace: 2 1 "+Integer.toHexString(10)+" "+
                             Long.toHexString(100)).putRecord(wtr);

      new Comment("S2_Two.putMetadata(wtr, S2_Two.A2, MetaTFConstants.Default, 333); "+
                  "trace: ! 2 2 1 "+Long.toHexString(333)).putRecord(wtr);
      S2_Two.putMetadata(wtr, S2_Two.A2, MetaTFConstants.Default, 333);
      new S2_Two(1, 10, 100, "S2_Two(1, 10, 100, cmt).putRecord(wtr); "+
```

```
            "trace: 2 1 "+Long.toHexString(100)).putRecord(wtr);
new S2_Two(1, 20, 100, "S2_Two(1, 20, 100, cmt).putRecord(wtr); "+
            "trace: 2 1 "+Long.toHexString(100)).putRecord(wtr);

new Comment("S2_Two.putMetadata(wtr, S2_Two.A2, MetaTFConstants.BaseOffset, 100); "+
             "trace: ! 2 2 0 "+Long.toHexString(100)).putRecord(wtr);
S2_Two.putMetadata(wtr, S2_Two.A2, MetaTFConstants.BaseOffset, 100);
new S2_Two(1, 110, 100, "S2_Two(1, 110, 100,cmt).putRecord(wtr); "+
            "trace: 2 1 a "+Long.toHexString(100)).putRecord(wtr);
new S2_Two(1, 111, 100, "S2_Two(1, 111, 100,cmt).putRecord(wtr); "+
            "trace: 2 1 b "+Long.toHexString(100)).putRecord(wtr);
new S2_Two(1, 112, 100, "S2_Two(1, 112, 100,cmt).putRecord(wtr); "+
            "trace: 2 1 c "+Long.toHexString(100)).putRecord(wtr);

new Comment("S2_Two.putMetadata(wtr, S2_Two.A2, MetaTFConstants.Delta, 50); "+
             "trace: ! 2 2 2 "+Long.toHexString(50)).putRecord(wtr);
S2_Two.putMetadata(wtr, S2_Two.A2, MetaTFConstants.Delta, 50);
new S2_Two(1, 50, 100, "S2_Two(1, 50, 100,cmt).putRecord(wtr); "+
            "trace: 2 1 0 "+Long.toHexString(100)).putRecord(wtr);
new S2_Two(1, 55, 100, "S2_Two(1, 55, 100,cmt).putRecord(wtr); "+
            "trace: 2 1 5 "+Long.toHexString(100)).putRecord(wtr);
new S2_Two(1, 55, 100, "S2_Two(1, 55, 100,cmt).putRecord(wtr); "+
            "trace: 2 1 0 "+Long.toHexString(100)).putRecord(wtr);
new S2_Two(1, 75, 100, "S2_Two(1, 75, 100,cmt).putRecord(wtr); "+
            "trace: 2 1 14 "+Long.toHexString(100)).putRecord(wtr);
new S2_Two(1, 65, 100, "S2_Two(1, 65, 100,cmt).putRecord(wtr); "+
            "trace: 2 1 -a "+Long.toHexString(100)).putRecord(wtr);

new Comment("S2_Two.putMetadata(wtr, S2_Two.A2, MetaTFConstants.Stride, 10, 20); "+
             "trace: ! 2 2 6 a "+Long.toHexString(20)).putRecord(wtr);
S2_Two.putMetadata(wtr, S2_Two.A2, MetaTFConstants.Stride, 10, 20);
new S2_Two(1, 2, 3, "S2_Two(1, 2, 3,cmt).putRecord(wtr); "+
            "trace: 2 1 3").putRecord(wtr);
new S2_Two(1, 2, 3, "S2_Two(1, 2, 3,cmt).putRecord(wtr); "+
            "trace: 2 1 3").putRecord(wtr);
new S2_Two(1, 2, 3, "S2_Two(1, 2, 3,cmt).putRecord(wtr); "+
            "trace: 2 1 3").putRecord(wtr);
new S2_Two(1, 2, 3, "S2_Two(1, 2, 3,cmt).putRecord(wtr); "+
            "trace: 2 1 3").putRecord(wtr);

S2_Three[] threes = new S2_Three[20];
for(int i=0; i<20; i++) {
  threes[i] = new S2_Three(0, new Index(ones[i], i),
                "S2_Three(0, new Index(ones["+i+"], "+i+"), cmt).putRecord(wtr); "+
                "trace: 3 0 "+Integer.toHexString(i));
  threes[i].putRecord(wtr);
}

new S2_Two(1, 2, 3, "S2_Two(1, 2, 3, cmt).putRecord(wtr); "+
            "trace: 2 1 3").putRecord(wtr);
new S2_Two(5, 6, 4, "S2_Two(5, 6, 4, cmt).putRecord(wtr); "+
            "trace: 2 5 4").putRecord(wtr);
new Comment("S2.putMetadata(wtr, Test.ALL_A, MetaTFConstants.Default, 5); "+
            "trace: ! * 2 1 5").putRecord(wtr);
S2.putMetadata(wtr, Test.ALL_A, MetaTFConstants.Default, 5);
new S2_Two(1, 2, 3, "S2_Two(1, 2, 3, cmt).putRecord(wtr); "+
            "trace: 2 3").putRecord(wtr);
new S2_Two(5, 6, 4, "S2_Two(5, 6, 4, cmt).putRecord(wtr); "+
```

```
                "trace: 2 4").putRecord(wtr);

      wtr.flush();
      wtr.close();

      // Now read this data back

      LowLevelReader rdr = new AsciiReader(new FileInputStream("test.trace"));
      Test.init();
      int i = 0;
      do {
        try {
          Record r = Test.getNextRecord(rdr);

          if(r instanceof Comment) {
            System.out.println("Comment: "+((Comment)r).text);
          } else if(r instanceof S1_One) {
            System.out.println("S1_One: "+((S1_One)r).desc);
          } else if(r instanceof S2_Two) {
            S2_Two r_ = (S2_Two)r;
            System.out.println("S2_Two: "+r_.a+" "+r_.a2+" "+r_.b+" "+r_.cmt);
          } else if(r instanceof S2_Three) {
                S2_Three r_ = (S2_Three)r;
                System.out.println("S2_Three: "+r_.a+"("+((S1_One)r_.b.record).desc+", "+
                                   r_.b.index+") "+r_.cmt);
          }
          i++;
        } catch (BadRecordException e) {
          System.err.println(">>>"+e.getMessage()+" at record "+i);
        } catch (IOException e) {
          System.err.println(i+" records read");
          System.exit(0);
        }
      } while(true);

    } catch(BadRecordException e) {
      System.err.println(e.getMessage());
    } catch(IOException e) {
      System.err.println(e.getMessage());
    }
  }
}
```