— THE —

Common Trace Format

A FLEXIBLE, HIGH-PERFORMANCE
BINARY TRACE FORMAT

The **Common Trace Format** (CTF) is a binary trace format designed to be *very fast to write* without compromising *great flexibility*. It allows traces to be natively generated by any C/C++ application or system, as well as by bare-metal (hardware) components.

With CTF, *all* headers, contexts, and event fields written in binary files are described using a custom C-like, declarative language called the *Trace Stream Description Language* (TSDL). Numerous binary trace stream layouts may be described in TSDL thanks to CTF's extensive range of available field types.

Babeltrace is the reference implementation of the Common Trace Format. It is a trace conversion application/C library which is able to read and write CTF, supporting almost all its specified features. Babeltrace also ships with Python 3 bindings to make it easier to open a CTF trace and iterate on its events in seconds.

This is the official documentation of **CTF v1.8.2**. Use the top right menu to select a different version.

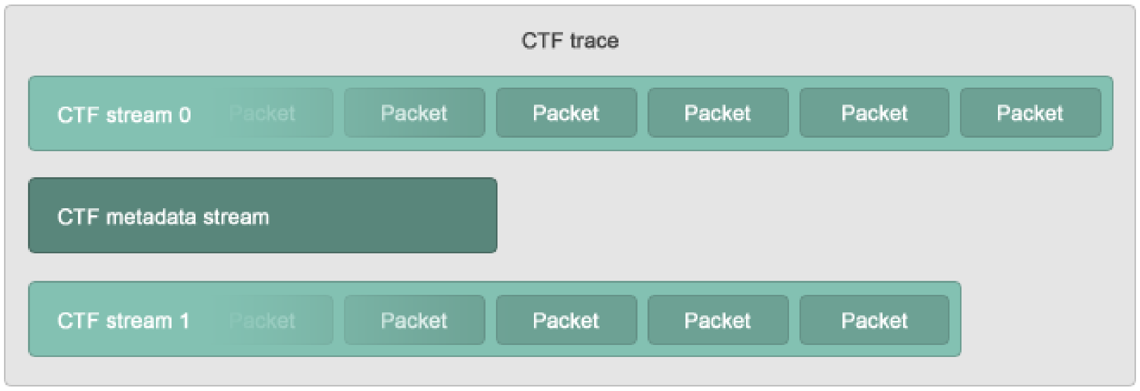
CTF IN A NUTSHELL

A CTF *trace* is composed of multiple *streams* of binary *events*. You are free to divide the events generated by your tracer into any number of different streams: since events need to be serialized in ascending order of timestamps, CTF readers are easily and efficiently able to flatten the events of multiple streams as an ordered list. For example, LTTng, a Linux kernel and user space tracer which outputs CTF traces natively, divides its events into one stream per CPU.

CTF does not require its streams to be actual files. CTF streams may as well be received/sent over the network, for example, and parsed in memory, without any data ever being written to disk.

One of a CTF trace's streams is mandatory: the *metadata* stream. It contains exactly what you would expect: data about the trace itself. The metadata stream contains a textual description of the binary layouts of all the other streams. This description is written using the *Trace Stream Description Language* (TSDL), a declarative language that exists only in the realm of CTF. The purpose of the metadata stream is to make CTF readers know how to parse a trace's binary streams of events without CTF specifying any fixed layout. The only stream layout known in advance is, in fact, the metadata stream's one.

A CTF binary stream is a concatenation of multiple *packets*.

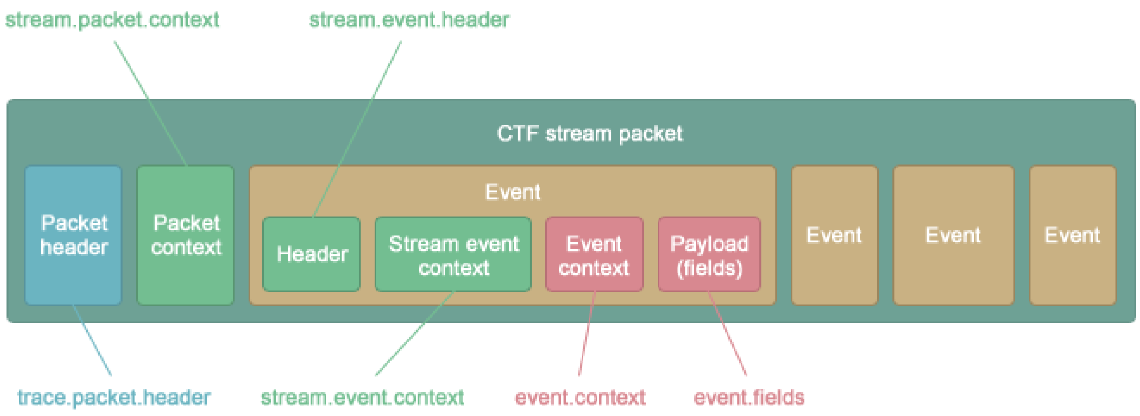


A CTF trace is composed of multiple CTF streams, one of which is the metadata stream. CTF streams are made of one or more packets.

A stream packet contains, in order:

1. A header
2. A context (optional)
3. Zero or more concatenated events, each containing:
    a. A header
    b. A stream-specific context (optional)
    c. An event-specific context (optional)
    d. A payload

Voilà! Padding may also exist between all those binary blocks, and between packets themselves.



All the stream headers, contexts and payloads are described in TSDL using CTF types, amongst:

- **Integers** of any size, any alignment
- **Floating point numbers** with any exponent and mantissa sizes
- Null-terminated **strings** of bytes
- **Enumerations** with ranges of integers mapped to labels
- Static and dynamic **arrays** of any CTF type
- **Structures** associating field names to any CTF types
- **Variants**, i.e. dynamic selections between different CTF types

This rich set of configurable types makes it possible to describe about any binary structure, hence CTF's great flexibility. On the other hand, this binary data is very fast to write for an application, as it's usually just a matter of appending some memory contents as is to a CTF stream.

SPECIFICATION

Common Trace Format (CTF) Specification (v1.8.2)

Author: Mathieu Desnoyers, EfficiOS Inc.

The goal of the present document is to specify a trace format that suits the needs of the embedded, telecom, high-performance and kernel communities. It is based on the Common Trace Format Requirements (v1.4) document. It is designed to allow traces to be natively generated by the Linux kernel, Linux user space applications written in C/C++, and hardware components. One major element of CTF is the Trace Stream Description Language (TSDL) which flexibility enables description of various binary trace stream layouts.

The latest version of this document can be found at:

- Git: `git clone git://git.efficios.com/ctf.git`
- Gitweb

A reference implementation of a library to read and write this trace format is being implemented within the Babeltrace project, a converter between trace formats. The development tree is available at:

- Git: `git clone git://git.efficios.com/babeltrace.git`
- Gitweb

The CE Workgroup of the Linux Foundation, Ericsson, and EfficiOS have sponsored this work.

Contents:

1. Preliminary definitions

- **Event trace**: an ordered sequence of events.
- **Event stream**: an ordered sequence of events, containing a subset of the trace event types.
- **Event packet**: a sequence of physically contiguous events within an event stream.
- **Event**: this is the basic entry in a trace. Also known as a *trace record*.
    - An **event identifier** (ID) relates to the class (a type) of event within an event stream, e.g. event `irq_entry`.
    - An **event** (or event record) relates to a specific instance of an event class, e.g. event `irq_entry`, at time *X*, on CPU *Y*.
- Source architecture: architecture writing the trace.
- Reader architecture: architecture reading the trace.

2. High-level representation of a trace

A *trace* is divided into multiple event streams. Each event stream contains a subset of the trace event types.

The final output of the trace, after its generation and optional transport over the network, is expected to be either on permanent or temporary storage in a virtual file system. Because each event stream is appended to while a trace is being recorded, each is associated with a distinct set of files for output. Therefore, a stored trace can be represented as a directory containing zero, one or more files per stream.

Metadata description associated with the trace contains information on trace event types expressed in the *Trace Stream Description Language* (TSDL). This language describes:

- Trace version
- Types available
- Per-trace event header description
- Per-stream event header description
- Per-stream event context description
- Per-event
    - Event type to stream mapping
    - Event type to name mapping
    - Event type to ID mapping
    - Event context description
    - Event fields description

3. Event stream

An *event stream* can be divided into contiguous event packets of variable size. An event packet can contain a certain amount of padding at the end. The stream header is repeated at the beginning of each event packet. The rationale for the event stream design choices is explained in Stream header rationale.

The event stream header will therefore be referred to as the *event packet header* throughout the rest of this document.

4. Types

Types are organized as type classes. Each type class belong to either of two kind of types: *basic types* or *compound types*.

4.1 Basic types

A basic type is a scalar type, as described in this section. It includes integers, GNU/C bitfields, enumerations, and floating point values.

4.1.1 TYPE INHERITANCE

Type specifications can be inherited to allow deriving types from a type class. For example, see the uint32_t named type derived from the *integer* type class. Types have a precise binary representation in the trace. A type class has methods to read and write these types, but must be derived into a type to be usable in an event field.

4.1.2 ALIGNMENT

We define *byte-packed* types as aligned on the byte size, namely 8-bit. We define *bit-packed* types as following on the next bit, as defined by the Integers section.

Each basic type must specify its alignment, in bits. Examples of possible alignments are: bit-packed (`align = 1`), byte-packed (`align = 8`), or word-aligned (e.g. `align = 32` or `align = 64`). The choice depends on the architecture preference and compactness vs performance trade-offs of the implementation. Architectures providing fast unaligned write byte-packed basic types to save space, aligning each type on byte boundaries (8-bit). Architectures with slow unaligned writes align types on specific alignment values. If no specific alignment is declared for a type, it is assumed to be bit-packed for integers with size not multiple of 8 bits and for gcc bitfields. All other basic types are byte-packed by default. It is however recommended to always specify the alignment explicitly. Alignment values must be power of two. Compound types are aligned as specified in their individual specification.

The base offset used for field alignment is the start of the packet containing the field. For instance, a field aligned on 32-bit needs to be at an offset multiple of 32-bit from the start of the packet that contains it.

TSDL metadata attribute representation of a specific alignment:

```
align = /* value in bits */;
```

4.1.3 **BYTE ORDER**

By default, byte order of a basic type is the byte order described in the trace description. It can be overridden by specifying a `byte_order` attribute for a basic type. Typical use-case is to specify the network byte order (big endian: `be`) to save data captured from the network into the trace without conversion.

TSDL metadata representation:

```
/* network and be are aliases */
byte_order = /* native OR network OR be OR le */;
```

The `native` keyword selects the byte order described in the trace description. The `network` byte order is an alias for big endian.

Even though the trace description section is not per se a type, for sake of clarity, it should be noted that `native` and `network` byte orders are only allowed within type declaration. The `byte_order` specified in the trace description section only accepts `be` or `le` values.

4.1.4 **SIZE**

Type size, in bits, for integers and floats is that returned by `sizeof()` in C multiplied by `CHAR_BIT` . We require the size of `char` and `unsigned char` types (`CHAR_BIT`) to be fixed to 8 bits for cross-endianness compatibility.

TSDL metadata representation:

```
size = /* value is in bits */;
```

4.1.5 **INTEGERS**

Signed integers are represented in two-complement. Integer alignment, size, signedness and byte ordering are defined in the TSDL metadata. Integers aligned on byte size (8-bit) and with length multiple of byte size (8-bit) correspond to the C99 standard integers. In addition, integers with alignment and/or size that are *not* a multiple of the byte size are permitted; these correspond to the C99 standard bitfields, with the added specification that the CTF integer bitfields have a fixed binary representation. Integer size needs to be a positive integer. Integers of size 0 are **forbidden**. An MIT-licensed reference implementation of the CTF portable bitfields is available here.

Binary representation of integers:

- On little and big endian:
  - Within a byte, high bits correspond to an integer high bits, and low bits correspond to low bits

- On little endian:
    - Integer across multiple bytes are placed from the less significant to the most significant
    - Consecutive integers are placed from lower bits to higher bits (even within a byte)
- On big endian:
    - Integer across multiple bytes are placed from the most significant to the less significant
    - Consecutive integers are placed from higher bits to lower bits (even within a byte)

This binary representation is derived from the bitfield implementation in GCC for little and big endian. However, contrary to what GCC does, integers can cross units boundaries (no padding is required). Padding can be explicitly added to follow the GCC layout if needed.

TSDL metadata representation:

```
integer {
    signed = /* true OR false */;                    /* default: false */
    byte_order = /* native OR network OR be OR le */; /* default: native */
    size = /* value in bits */;                      /* no default */
    align = /* value in bits */;

    /* base used for pretty-printing output; default: decimal */
    base = /* decimal OR dec OR d OR i OR u OR 10 OR hexadecimal OR hex
              OR x OR X OR p OR 16 OR octal OR oct OR o OR 8 OR binary
              OR b OR 2 */;

    /* character encoding */
    encoding = /* none or UTF8 or ASCII */;          /* default: none */
}
```

Example of type inheritance (creation of a `uint32_t` named type):

```
typealias integer {
    size = 32;
    signed = false;
    align = 32;
} := uint32_t;
```

Definition of a named 5-bit signed bitfield:

```
typealias integer {
    size = 5;
    signed = true;
    align = 1;
} := int5_t;
```

The character encoding field can be used to specify that the integer must be printed as a text character when read. e.g.:

```
typealias integer {
    size = 8;
    align = 8;
    signed = false;
    encoding = UTF8;
} := utf_char;
```

4.1.6 **GNU/C BITFIELDS**

The GNU/C bitfields follow closely the integer representation, with a particularity on alignment: if a bitfield cannot fit in the current unit, the unit is padded and the bitfield starts at the following unit. The unit size is defined by the size of the type `unit_type`.

TSDL metadata representation:

```
unit_type name:size;
```

As an example, the following structure declared in C compiled by GCC:

```
struct example {
    short a:12;
    short b:5;
};
```

The example structure is aligned on the largest element (short). The second bitfield would be aligned on the next unit boundary, because it would not fit in the current unit.

4.1.7 **FLOATING POINT**

The floating point values byte ordering is defined in the TSDL metadata.

Floating point values follow the IEEE 754-2008 standard interchange formats. Description of the floating point values include the exponent and mantissa size in bits. Some requirements are imposed on the floating point values:

- `FLT_RADIX` must be 2.
- `mant_dig` is the number of digits represented in the mantissa. It is specified by the ISO C99 standard, section 5.2.4, as `FLT_MANT_DIG`, `DBL_MANT_DIG` and `LDBL_MANT_DIG` as defined by `<float.h>`.
- `exp_dig` is the number of digits represented in the exponent. Given that `mant_dig` is one bit more than its actual size in bits (leading 1 is not needed) and also given that the sign bit always takes one bit, `exp_dig` can be specified as:
  - `sizeof(float) * CHAR_BIT - FLT_MANT_DIG`
  - `sizeof(double) * CHAR_BIT - DBL_MANT_DIG`
  - `sizeof(long double) * CHAR_BIT - LDBL_MANT_DIG`

TSDL metadata representation:

```
floating_point {
    exp_dig = /* value */;
    mant_dig = /* value */;
    byte_order = /* native OR network OR be OR le */;
    align = /* value */;
}
```

Example of type inheritance:

```
typealias floating_point {
    exp_dig = 8;         /* sizeof(float) * CHAR_BIT - FLT_MANT_DIG */
    mant_dig = 24;       /* FLT_MANT_DIG */
    byte_order = native;
    align = 32;
} := float;
```

TODO: define NaN, +inf, -inf behavior.

Bit-packed, byte-packed or larger alignments can be used for floating point values, similarly to integers.

4.1.8 **ENUMERATIONS**

Enumerations are a mapping between an integer type and a table of strings. The numerical representation of the enumeration follows the integer type specified by the metadata. The enumeration mapping table is detailed in the enumeration description within the metadata. The mapping table maps inclusive value ranges (or single values) to strings. Instead of being limited to simple `value -> string` mappings, these enumerations map `[ start_value ... end_value ] -> string`, which map inclusive ranges of values to strings. An enumeration from the C language can be represented in this format by having the same `start_value` and `end_value` for each mapping, which is in fact a range of size 1. This single-value range is supported without repeating the start and end values with the `value = string` declaration. Enumerations need to contain at least one entry.

```
enum name : integer_type {
    somestring          = /* start_value1 */ ... /* end_value1 */,
    "other string"      = /* start_value2 */ ... /* end_value2 */,
    yet_another_string,   /* will be assigned to end_value2 + 1 */
    "some other string" = /* value */,
    /* ... */
}
```

If the values are omitted, the enumeration starts at 0 and increment of 1 for each entry. An entry with omitted value that follows a range entry takes as value the `end_value` of the previous range + 1:

```
enum name : unsigned int {
    ZERO,
    ONE,
    TWO,
    TEN = 10,
    ELEVEN,
}
```

Overlapping ranges within a single enumeration are implementation defined.

A nameless enumeration can be declared as a field type or as part of a `typedef`:

```
enum : integer_type {
    /* ... */
}
```

Enumerations omitting the container type `: integer_type` use the `int` type (for compatibility with C99). The `int` type *must be* previously declared, e.g.:

```
typealias integer { size = 32; align = 32; signed = true; } := int;

enum {
    /* ... */
}
```

4.2 Compound types

Compound are aggregation of type declarations. Compound types include structures, variant, arrays, sequences, and strings.

4.2.1 STRUCTURES

Structures are aligned on the largest alignment required by basic types contained within the structure. (This follows the ISO/C standard for structures)

TSDL metadata representation of a named structure:

```
struct name {
    field_type field_name;
    field_type field_name;
    /* ... */
};
```

Example:

```
struct example {
    integer {                   /* nameless type */
        size = 16;
        signed = true;
        align = 16;
    } first_field_name;
    uint64_t second_field_name; /* named type declared in the metadata */
};
```

The fields are placed in a sequence next to each other. They each possess a field name, which is a unique identifier within the structure. The identifier is not allowed to use any reserved keyword. Replacing reserved keywords with underscore-prefixed field names is **recommended**.

Fields starting with an underscore should have their leading underscore removed by the CTF trace readers.

A nameless structure can be declared as a field type or as part of a `typedef`:

```
struct {
    /* ... */
}
```

Alignment for a structure compound type can be forced to a minimum value by adding an `align` specifier after the declaration of a structure body. This attribute is read as: `align(value)`. The value is specified in bits. The structure will be aligned on the maximum value between this attribute and the alignment required by the basic types contained within the structure. e.g.

```
struct {
    /* ... */
} align(32)
```

4.2.2 **VARIANTS (DISCRIMINATED/TAGGED UNIONS)**

A CTF variant is a selection between different types. A CTF variant must always be defined within the scope of a structure or within fields contained within a structure (defined recursively). A *tag* enumeration field must appear in either the same static scope, prior to the variant field (in field declaration order), in an upper static scope, or in an upper dynamic scope (see Static and dynamic scopes). The type selection is indicated by the mapping from the enumeration value to the string used as variant type selector. The field to use as tag is specified by the `tag_field`, specified between `< >` after the `variant` keyword for unnamed variants, and after *variant name* for named variants. It is not required that each enumeration mapping appears as variant type tag field. It is also not required that each variant type tag appears as enumeration mapping. However, it is required that any enumeration mapping encountered within a stream has a matching variant type tag field.

The alignment of the variant is the alignment of the type as selected by the tag value for the specific instance of the variant. The size of the variant is the size as selected by the tag value for the specific instance of the variant.

The alignment of the type containing the variant is independent of the variant alignment. For instance, if a structure contains two fields, a 32-bit integer, aligned on 32 bits, and a variant, which contains two choices: either a 32-bit field, aligned on 32 bits, or a 64-bit field, aligned on 64 bits, the alignment of the outmost structure will be 32-bit (the alignment of its largest field, disregarding the alignment of the variant). The alignment of the variant will depend on the selector: if the variant's 32-bit field is selected, its alignment will be 32-bit, or 64-bit otherwise. It is important to note that variants are specifically tailored for compactness in a stream. Therefore, the relative offsets of compound type fields can vary depending on the offset at which the compound type starts if it contains a variant that itself contains a type with alignment larger than the largest field contained within the compound type. This is caused by the fact that the compound type may contain the enumeration that select the variant's choice, and therefore the alignment to be applied to the compound type cannot be determined before encountering the enumeration.

Each variant type selector possess a field name, which is a unique identifier within the variant. The identifier is not allowed to use any reserved keyword. Replacing reserved keywords with underscore-prefixed field names is recommended. Fields starting with an underscore should have their leading underscore removed by the CTF trace readers.

A named variant declaration followed by its definition within a structure declaration:

```
variant name {
    field_type sel1;
    field_type sel2;
    field_type sel3;
    /* ... */
};

struct {
    enum : integer_type { sel1, sel2, sel3, /* ... */ } tag_field;
    /* ... */
    variant name <tag_field> v;
}
```

An unnamed variant definition within a structure is expressed by the following TSDL metadata:

```
struct {
    enum : integer_type { sel1, sel2, sel3, /* ... */ } tag_field;
    /* ... */
    variant <tag_field> {
        field_type sel1;
        field_type sel2;
        field_type sel3;
        /* ... */
    } v;
}
```

Example of a named variant within a sequence that refers to a single tag field:

```
variant example {
    uint32_t a;
    uint64_t b;
    short c;
};

struct {
    enum : uint2_t { a, b, c } choice;
    unsigned int seqlen;
    variant example <choice> v[seqlen];
}
```

Example of an unnamed variant:

```
struct {
    enum : uint2_t { a, b, c, d } choice;

    /* Unrelated fields can be added between the variant and its tag */
    int32_t somevalue;
    variant <choice> {
        uint32_t a;
        uint64_t b;
        short c;
        struct {
            unsigned int field1;
            uint64_t field2;
        } d;
    } s;
}
```

Example of an unnamed variant within an array:

```
struct {
    enum : uint2_t { a, b, c } choice;
    variant <choice> {
        uint32_t a;
        uint64_t b;
        short c;
    } v[10];
}
```

Example of a variant type definition within a structure, where the defined type is then declared within an array of structures. This variant refers to a tag located in an upper static scope. This example clearly shows that a variant type definition referring to the tag `x` uses the closest preceding field from the static scope of the type definition.

```
struct {
    enum : uint2_t { a, b, c, d } x;

    /*
     * "x" refers to the preceding "x" enumeration in the
     * static scope of the type definition.
     */
    typedef variant <x> {
      uint32_t a;
      uint64_t b;
      short c;
    } example_variant;

    struct {
      enum : int { x, y, z } x; /* This enumeration is not used by "v". */

      /* "v" uses the "enum : uint2_t { a, b, c, d }" tag. */
      example_variant v;
    } a[10];
}
```

4.2.3 **ARRAYS**

Arrays are fixed-length. Their length is declared in the type declaration within the metadata. They contain an array of *inner type* elements, which can refer to any type not containing the type of the array being declared (no circular dependency). The length is the number of elements in an array.

TSDL metadata representation of a named array:

```
typedef elem_type name[/* length */];
```

A nameless array can be declared as a field type within a structure, e.g.:

```
uint8_t field_name[10];
```

Arrays are always aligned on their element alignment requirement.

4.2.4 **SEQUENCES**

Sequences are dynamically-sized arrays. They refer to a *length* unsigned integer field, which must appear in either the same static scope, prior to the sequence field (in field declaration order), in an upper static scope, or in an upper dynamic scope (see Static and dynamic scopes). This length field represents the number of elements in the sequence. The sequence per se is an array of *inner type* elements.

TSDL metadata representation for a sequence type definition:

```
struct {
    unsigned int length_field;
    typedef elem_type typename[length_field];
    typename seq_field_name;
}
```

A sequence can also be declared as a field type, e.g.:

```
struct {
    unsigned int length_field;
    long seq_field_name[length_field];
}
```

Multiple sequences can refer to the same length field, and these length fields can be in a different upper dynamic scope, e.g., assuming the `stream.event.header` defines:

```
stream {
    /* ... */
    id = 1;
    event.header := struct {
        uint16_t seq_len;
    };
};

event {
    /* ... */
    stream_id = 1;
    fields := struct {
        long seq_a[stream.event.header.seq_len];
        char seq_b[stream.event.header.seq_len];
    };
};
```

The sequence elements follow the array specifications.

4.2.5 STRINGS

Strings are an array of *bytes* of variable size and are terminated by a `'\0'` “NULL” character. Their encoding is described in the TSDL metadata. In absence of encoding attribute information, the default encoding is UTF-8.

TSDL metadata representation of a named string type:

```
typealias string {
    encoding = /* UTF8 OR ASCII */;
} := name;
```

A nameless string type can be declared as a field type:

```
string field_name; /* use default UTF8 encoding */
```

Strings are always aligned on byte size.

5. Event packet header

The event packet header consists of two parts: the *event packet header* is the same for all streams of a trace. The second part, the *event packet context*, is described on a per-stream basis. Both are described in the TSDL metadata.

Event packet header (all fields are optional, specified by TSDL metadata):

- **Magic number** (CTF magic number: 0xC1FC1FC1) specifies that this is a CTF packet. This magic number is optional, but when present, it should come at the very beginning of the packet.
- **Trace UUID**, used to ensure the event packet match the metadata used. Note: we cannot use a metadata checksum in every cases instead of a UUID because metadata can be appended to while tracing is active. This field is optional.
- **Stream ID**, used as reference to stream description in metadata. This field is optional if there is only one stream description in the metadata, but becomes required if there are more than one stream in the TSDL metadata description.

Event packet context (all fields are optional, specified by TSDL metadata):

- Event packet **content size** (in bits).
- Event packet **size** (in bits, includes padding).
- Event packet content checksum. Checksum excludes the event packet header.
- Per-stream event **packet sequence count** (to deal with UDP packet loss). The number of significant sequence counter bits should also be present, so wrap-arounds are dealt with

correctly.

- Time-stamp at the beginning and timestamp at the end of the event packet. Both timestamps are written in the packet header, but sampled respectively while (or before) writing the first event and while (or after) writing the last event in the packet. The inclusive range between these timestamps should include all event timestamps assigned to events contained within the packet. The timestamp at the beginning of an event packet is guaranteed to be below or equal the timestamp at the end of that event packet. The timestamp at the end of an event packet is guaranteed to be below or equal the timestamps at the end of any following packet within the same stream. See Clocks for more detail.
- **Events discarded count**. Snapshot of a per-stream free-running counter, counting the number of events discarded that were supposed to be written in the stream after the last event in the event packet. Note: producer-consumer buffer full condition can fill the current event packet with padding so we know exactly where events have been discarded. However, if the buffer full condition chooses not to fill the current event packet with padding, all we know about the timestamp range in which the events have been discarded is that it is somewhere between the beginning and the end of the packet.
- Lossless **compression scheme** used for the event packet content. Applied directly to raw data. New types of compression can be added in following versions of the format.
  - 0: no compression scheme
  - 1: bzip2
  - 2: gzip
  - 3: xz
- **Cypher** used for the event packet content. Applied after compression.
  - 0: no encryption
  - 1: AES
- **Checksum scheme** used for the event packet content. Applied after encryption.
  - 0: no checksum
  - 1: md5
  - 2: sha1
  - 3: crc32

5.1 Event packet header description

The event packet header layout is indicated by the `trace.packet.header` field. Here is a recommended structure type for the packet header with the fields typically expected (although these fields are each optional):

```
struct event_packet_header {
    uint32_t magic;
    uint8_t  uuid[16];
    uint32_t stream_id;
};

trace {
    /* ... */
    packet.header := struct event_packet_header;
};
```

If the magic number is not present, tools such as `file` will have no mean to discover the file type.

If the uuid is not present, no validation that the metadata actually corresponds to the stream is performed.

If the stream_id packet header field is missing, the trace can only contain a single stream. Its `id` field can be left out, and its events don't need to declare a `stream_id` field.

5.2 **Event packet context description**

Event packet context example. These are declared within the stream declaration in the metadata. All these fields are optional. If the packet size field is missing, the whole stream only contains a single packet. If the content size field is missing, the packet is filled (no padding). The content and packet sizes include all headers.

An example event packet context type:

```
struct event_packet_context {
    uint64_t timestamp_begin;
    uint64_t timestamp_end;
    uint32_t checksum;
    uint32_t stream_packet_count;
    uint32_t events_discarded;
    uint32_t cpu_id;
    uint64_t content_size;
    uint64_t packet_size;
    uint8_t  compression_scheme;
    uint8_t  encryption_scheme;
    uint8_t  checksum_scheme;
};
```

6. Event Structure

The overall structure of an event is:

1. Event header (as specified by the stream metadata)
2. Stream event context (as specified by the stream metadata)
3. Event context (as specified by the event metadata)
4. Event payload (as specified by the event metadata)

This structure defines an implicit dynamic scoping, where variants located in inner structures (those with a higher number in the listing above) can refer to the fields of outer structures (with lower number in the listing above). See TSDL scopes for more detail.

The total length of an event is defined as the difference between the end of its event payload and the end of the previous event's event payload. Therefore, it includes the event header alignment padding, and all its fields and their respective alignment padding. Events of length 0 are forbidden.

6.1 **Event header**

Event headers can be described within the metadata. We hereby propose, as an example, two types of events headers. Type 1 accommodates streams with less than 31 event IDs. Type 2 accommodates streams with 31 or more event IDs.

One major factor can vary between streams: the number of event IDs assigned to a stream. Luckily, this information tends to stay relatively constant (modulo event registration while trace is being recorded), so we can specify different representations for streams containing few event IDs and streams containing many event IDs, so we end up representing the event ID and timestamp as densely as possible in each case.

The header is extended in the rare occasions where the information cannot be represented in the ranges available in the standard event header. They are also used in the rare occasions where the data required for a field could not be collected: the flag corresponding to the missing field within the `missing_fields` array is then set to 1.

Types `uintX_t` represent an `X`-bit unsigned integer, as declared with either:

```
typealias integer {
    size = /* X */;
    align = /* X */;
    signed = false;
} := uintX_t;
```

or

```
typealias integer {
    size = /* X */;
    align = 1;
    signed = false;
} := uintX_t;
```

For more information about timestamp fields, see Clocks.

6.1.1 **TYPE 1: FEW EVENT IDS**

- Aligned on 32-bit (or 8-bit if byte-packed, depending on the architecture preference)
- Native architecture byte ordering
- For `compact` selection, fixed size of 32 bits
- For "extended" selection, size depends on the architecture and variant alignment

```
struct event_header_1 {
    /*
     * id: range: 0 - 30.
     * id 31 is reserved to indicate an extended header.
     */
    enum : uint5_t { compact = 0 ... 30, extended = 31 } id;
    variant <id> {
        struct {
            uint27_t timestamp;
        } compact;
        struct {
            uint32_t id;        /* 32-bit event IDs */
            uint64_t timestamp; /* 64-bit timestamps */
        } extended;
    } v;
} align(32); /* or align(8) */
```

6.1.2 **TYPE 2: MANY EVENT IDS**

- Aligned on 16-bit (or 8-bit if byte-packed, depending on the architecture preference)
- Native architecture byte ordering
- For `compact` selection, size depends on the architecture and variant alignment
- For `extended` selection, size depends on the architecture and variant alignment

```
struct event_header_2 {
    /*
     * id: range: 0 - 65534.
     * id 65535 is reserved to indicate an extended header.
     */
    enum : uint16_t { compact = 0 ... 65534, extended = 65535 } id;
    variant <id> {
        struct {
            uint32_t timestamp;
        } compact;
        struct {
            uint32_t id;        /* 32-bit event IDs */
            uint64_t timestamp; /* 64-bit timestamps */
        } extended;
    } v;
} align(16); /* or align(8) */
```

6.2 **Stream event context and event context**

The event context contains information relative to the current event. The choice and meaning of this information is specified by the TSDL stream and event metadata descriptions. The stream context is applied to all events within the stream. The stream context structure follows the event header. The event context is applied to specific events. Its structure follows the stream context structure.

An example of stream-level event context is to save the event payload size with each event, or to save the current PID with each event. These are declared within the stream declaration within the metadata:

```
stream {
    /* ... */
    event.context := struct {
        uint pid;
        uint16_t payload_size;
    };
};
```

An example of event-specific event context is to declare a bitmap of missing fields, only appended after the stream event context if the extended event header is selected. `NR_FIELDS` is the number of fields within the event (a numeric value).

```
event {
    context := struct {
        variant <id> {
            struct { } compact;
            struct {
                /* missing event fields bitmap */
                uint1_t missing_fields[NR_FIELDS];
            } extended;
        } v;
    };
    /* ... */
}
```

6.3 **Event payload**

An event payload contains fields specific to a given event type. The fields belonging to an event type are described in the event-specific metadata within a structure type.

6.3.1 **PADDING**

No padding at the end of the event payload. This differs from the ISO/C standard for structures, but follows the CTF standard for structures. In a trace, even though it makes sense to align the beginning of a structure, it really makes no sense to add padding at the end of the structure, because structures are usually not followed by a structure of the same type.

This trick can be done by adding a zero-length `end` field at the end of the C structures, and by using the offset of this field rather than using `sizeof()` when calculating the size of a structure (see Helper macros).

6.3.2 **ALIGNMENT**

The event payload is aligned on the largest alignment required by types contained within the payload. This follows the ISO/C standard for structures.

7. Trace Stream Description Language (TSDL)

The Trace Stream Description Language (TSDL) allows expression of the binary trace streams layout in a C99-like Domain Specific Language (DSL).

7.1 **Meta-data**

The trace stream layout description is located in the trace metadata. The metadata is itself located in a stream identified by its name: `metadata`.

The metadata description can be expressed in two different formats: text-only and packet-based. The text-only description facilitates generation of metadata and provides a convenient way to enter the metadata information by hand. The packet-based metadata provides the CTF stream packet facilities (checksumming, compression, encryption, network-readiness) for metadata stream generated and transported by a tracer.

The text-only metadata file is a plain-text TSDL description. This file must begin with the following characters to identify the file as a CTF TSDL text-based metadata file (without the double-quotes):

```
"/* CTF"
```

It must be followed by a space, and the version of the specification followed by the CTF trace, e.g.:

```
" 1.8"
```

These characters allow automated discovery of file type and CTF specification version. They are interpreted as a the beginning of a comment by the TSDL metadata parser. The comment can be continued to contain extra commented characters before it is closed.

The packet-based metadata is made of *metadata packets*, which each start with a metadata packet header. The packet-based metadata description is detected by reading the magic number 0x75D11D57 at the beginning of the file. This magic number is also used to detect the endianness of the architecture by trying to read the CTF magic number and its counterpart in reversed endianness. The events within the metadata stream have no event header nor event context. Each event only contains a special *sequence* payload, which is a sequence of bits which length is implicitly calculated by using the `trace.packet.header.content_size` field, minus the packet header size. The formatting of this sequence of bits is a plain-text representation of the TSDL description. Each metadata packet start with a special packet header, specific to the metadata stream, which contains, exactly:

```
struct metadata_packet_header {
    uint32_t magic;              /* 0x75D11D57 */
    uint8_t  uuid[16];           /* Unique Universal Identifier */
    uint32_t checksum;           /* 0 if unused */
    uint32_t content_size;       /* in bits */
    uint32_t packet_size;        /* in bits */
    uint8_t  compression_scheme; /* 0 if unused */
    uint8_t  encryption_scheme;  /* 0 if unused */
    uint8_t  checksum_scheme;    /* 0 if unused */
    uint8_t  major;              /* CTF spec version major number */
    uint8_t  minor;              /* CTF spec version minor number */
};
```

The packet-based metadata can be converted to a text-only metadata by concatenating all the strings it contains.

In the textual representation of the metadata, the text contained within `/*` and `*/`, as well as within `//` and end of line, are treated as comments. Boolean values can be represented as `true`, `TRUE`, or `1` for true, and `false`, `FALSE`, or `0` for false. Within the string-based metadata description, the trace UUID is represented as a string of hexadecimal digits and dashes `-`. In the event packet header, the trace UUID is represented as an array of bytes.

7.2 **Declaration vs definition**

A declaration associates a layout to a type, without specifying where this type is located in the event structure hierarchy. This therefore includes `typedef`, `typealias`, as well as all type specifiers. In certain circumstances (`typedef` , structure field and variant field), a declaration is followed by a declarator, which specify the newly defined type name (for `typedef`), or the field name (for declarations located within structure and variants). Array and sequence, declared with square brackets (`[ ]`), are part of the declarator, similarly to C99. The enumeration base type is specified by `: enum_base` , which is part of the type specifier. The variant tag name, specified between `< >` , is also part of the type specifier.

A definition associates a type to a location in the event structure hierarchy. This association is denoted by `:=` , as shown in TSDL scopes.

7.3 TSDL scopes

TSDL uses three different types of scoping: a lexical scope is used for declarations and type definitions, and static and dynamic scopes are used for variants references to tag fields (with relative and absolute path lookups) and for sequence references to length fields.

7.3.1 LEXICAL SCOPE

Each of `trace` , `env` , `stream` , `event` , `struct` and `variant` have their own nestable declaration scope, within which types can be declared using `typedef` and `typealias` . A root declaration scope also contains all declarations located outside of any of the aforementioned declarations. An inner declaration scope can refer to type declared within its container lexical scope prior to the inner declaration scope. Redefinition of a typedef or typealias is not valid, although hiding an upper scope typedef or typealias is allowed within a sub-scope.

7.3.2 STATIC AND DYNAMIC SCOPES

A local static scope consists in the scope generated by the declaration of fields within a compound type. A static scope is a local static scope augmented with the nested sub-static-scopes it contains.

A dynamic scope consists in the static scope augmented with the implicit event structure definition hierarchy.

Multiple declarations of the same field name within a local static scope is not valid. It is however valid to re-use the same field name in different local scopes.

Nested static and dynamic scopes form lookup paths. These are used for variant tag and sequence length references. They are used at the variant and sequence definition site to look up the location of the tag field associated with a variant, and to lookup up the location of the length field associated with a sequence.

Variants and sequences can refer to a tag field either using a relative path or an absolute path. The relative path is relative to the scope in which the variant or sequence performing the lookup is located. Relative paths are only allowed to lookup within the same static scope, which includes its nested static scopes. Lookups targeting parent static scopes need to be performed with an absolute path.

Absolute path lookups use the full path including the dynamic scope followed by a `.` and then the static scope. Therefore, variants (or sequences) in lower levels in the dynamic scope (e.g., event context) can refer to a tag (or length) field located in upper levels (e.g., in the event header) by specifying, in this case, the associated tag with `<stream.event.header.field_name>`. This allows, for instance, the event context to define a variant referring to the `id` field of the event header as selector.

The dynamic scope prefixes are thus:

- Trace environment: `<env. >`
- Trace packet header: `<trace.packet.header. >`
- Stream packet context: `<stream.packet.context. >`
- Event header: `<stream.event.header. >`
- Stream event context: `<stream.event.context. >`
- Event context: `<event.context. >`
- Event payload: `<event.fields. >`

The target dynamic scope must be specified explicitly when referring to a field outside of the static scope (absolute scope reference). No conflict can occur between relative and dynamic paths, because the keywords `trace`, `stream`, and `event` are reserved, and thus not permitted as field names. It is recommended that field names clashing with CTF and C99 reserved keywords use an underscore prefix to eliminate the risk of generating a description containing an invalid field name. Consequently, fields starting with an underscore should have their leading underscore removed by the CTF trace readers.

The information available in the dynamic scopes can be thought of as the current tracing context. At trace production, information about the current context is saved into the specified scope field levels. At trace consumption, for each event, the current trace context is therefore readable by accessing the upper dynamic scopes.

7.4 **TSDL examples**

The grammar representing the TSDL metadata is presented in TSDL grammar. This section presents a rather lighter reading that consists in examples of TSDL metadata, with template values.

The stream ID can be left out if there is only one stream in the trace. The event `id` field can be left out if there is only one event in a stream.

```
trace {
    major = /* value */;            /* CTF spec version major number */
    minor = /* value */;            /* CTF spec version minor number */
    uuid = "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa";  /* Trace UUID */
    byte_order = /* be OR le */;    /* Endianness (required) */
    packet.header := struct {
        uint32_t magic;
        uint8_t  uuid[16];
        uint32_t stream_id;
    };
};

/*
 * The "env" (environment) scope contains assignment expressions. The
 * field names and content are implementation-defined.
 */
env {
    pid = /* value */;    /* example */
    proc_name = "name";   /* example */
    /* ... */
};

stream {
    id = /* stream_id */;
    /* Type 1 - Few event IDs; Type 2 - Many event IDs. See section 6.1. */
    event.header := /* event_header_1 OR event_header_2 */;
    event.context := struct {
        /* ... */
    };
    packet.context := struct {
        /* ... */
    };
};

event {
    name = "event_name";
    id = /* value */;            /* Numeric identifier within the stream */
    stream_id = /* stream_id */;
    loglevel = /* value */;
    model.emf.uri = "string";
    context := struct {
        /* ... */
    };
    fields := struct {
        /* ... */
    };
};

callsite {
    name = "event_name";
    func = "func_name";
    file = "myfile.c";
    line = 39;
    ip = 0x40096c;
};
```

More detail on types:

```
/*
 * Named types:
 *
 * Type declarations behave similarly to the C standard.
 */

typedef aliased_type_specifiers new_type_declarators;

/* e.g.: typedef struct example new_type_name[10]; */

/*
 * typealias
 *
 * The "typealias" declaration can be used to give a name (including
 * pointer declarator specifier) to a type. It should also be used to
 * map basic C types (float, int, unsigned long, ...) to a CTF type.
 * Typealias is a superset of "typedef": it also allows assignment of a
 * simple variable identifier to a type.
 */

typealias type_class {
    /* ... */
} := type_specifiers type_declarator;

/*
 * e.g.:
 * typealias integer {
 *   size = 32;
 *   align = 32;
 *   signed = false;
 * } := struct page *;
 *
 * typealias integer {
 *   size = 32;
 *   align = 32;
 *   signed = true;
 * } := int;
 */

struct name {
    /* ... */
};

variant name {
    /* ... */
};

enum name : integer_type {
    /* ... */
};
```

Unnamed types, contained within compound type fields, `typedef` or `typealias`:

```
struct {
    /* ... */
}
```

```
struct {
    /* ... */
} align(value)
```

```
variant {
    /* ... */
}
```

```
enum : integer_type {
    /* ... */
}
```

```
typedef type new_type[length];

struct {
    type field_name[length];
}
```

```
typedef type new_type[length_type];

struct {
    type field_name[length_type];
}
```

```
integer {
    /* ... */
}
```

```
floating_point {
    /* ... */
}
```

```
struct {
    integer_type field_name:size;   /* GNU/C bitfield */
}
```

```
struct {
    string field_name;
}
```

8. Clocks

Clock metadata allows to describe the clock topology of the system, as well as to detail each clock parameter. In absence of clock description, it is assumed that all fields named `timestamp` use the same clock source, which increments once per nanosecond.

Describing a clock and how it is used by streams is threefold: first, the clock and clock topology should be described in a `clock` description block, e.g.:

```
clock {
    name = cycle_counter_sync;
    uuid = "62189bee-96dc-11e0-91a8-cfa3d89f3923";
    description = "Cycle counter synchronized across CPUs";
    freq = 1000000000;           /* frequency, in Hz */
    /* precision in seconds is: 1000 * (1/freq) */
    precision = 1000;
    /*
     * clock value offset from Epoch is:
     * offset_s + (offset * (1/freq))
     */
    offset_s = 1326476837;
    offset = 897235420;
    absolute = FALSE;
};
```

The mandatory `name` field specifies the name of the clock identifier, which can later be used as a reference. The optional field `uuid` is the unique identifier of the clock. It can be used to correlate different traces that use the same clock. An optional textual description string can be added with the `description` field. The `freq` field is the initial frequency of the clock, in Hz. If the `freq` field is not present, the frequency is assumed to be 1000000000 (providing clock increment of 1 ns). The optional `precision` field details the uncertainty on the clock measurements, in (1/freq) units. The `offset_s` and `offset` fields indicate the offset from POSIX.1 Epoch, 1970-01-01 00:00:00 +0000 (UTC), to the zero of value of the clock. The `offset_s` field is in seconds. The `offset` field is in (1/freq) units. If any of the `offset_s` or `offset` field is not present, it is assigned the 0 value. The field `absolute` is `TRUE` if the clock is a global reference across different clock UUID (e.g. NTP time). Otherwise, `absolute` is `FALSE`, and the clock can be considered as synchronized only with other clocks that have the same UUID.

Secondly, a reference to this clock should be added within an integer type:

```
typealias integer {
    size = 64; align = 1; signed = false;
    map = clock.cycle_counter_sync.value;
} := uint64_ccnt_t;
```

Thirdly, stream declarations can reference the clock they use as a timestamp source:

```
struct packet_context {
    uint64_ccnt_t ccnt_begin;
    uint64_ccnt_t ccnt_end;
    /* ... */
};

stream {
    /* ... */
    event.header := struct {
        uint64_ccnt_t timestamp;
        /* ... */
    };
    packet.context := struct packet_context;
};
```

For a N-bit integer type referring to a clock, if the integer overflows compared to the N low order bits of the clock prior value found in the same stream, then it is assumed that one, and only one, overflow occurred. It is therefore important that events encoding time on a small number of bits happen frequently enough to detect when more than one N-bit overflow occurs.

In a packet context, clock field names ending with `_begin` and `_end` have a special meaning: this refers to the timestamps at, respectively, the beginning and the end of each packet.

A. Helper macros

The two following macros keep track of the size of a GNU/C structure without padding at the end by placing HEADER_END as the last field. A one byte end field is used for C90 compatibility (C99 flexible arrays could be used here). Note that this does not affect the effective structure size, which should always be calculated with the `header_sizeof()` helper.

```
#define HEADER_END          char end_field
#define header_sizeof(type) offsetof(typeof(type), end_field)
```

B. Stream header rationale

An event stream is divided in contiguous event packets of variable size. These subdivisions allow the trace analyzer to perform a fast binary search by time within the stream (typically requiring to index only the event packet headers) without reading the whole stream. These subdivisions have a variable size to eliminate the need to transfer the event packet padding when partially filled event packets must be sent when streaming a trace for live viewing/analysis. An event packet can contain a certain amount of padding at the end. Dividing streams into event packets is also useful for network streaming over UDP and flight recorder mode tracing (a whole event packet can be swapped out of the buffer atomically for reading).

The stream header is repeated at the beginning of each event packet to allow flexibility in terms of:

- streaming support
- allowing arbitrary buffers to be discarded without making the trace unreadable
- allow UDP packet loss handling by either dealing with missing event packet or asking for re-transmission
- transparently support flight recorder mode
- transparently support crash dump

C. TSDL Grammar

```
/*
 * Common Trace Format (CTF) Trace Stream Description Language (TSDL) Grammar.
 *
 * Inspired from the C99 grammar:
 * http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf (Annex A)
 * and c++1x grammar (draft)
 * http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3291.pdf (Annex A)
 *
 * Specialized for CTF needs by including only constant and declarations from
 * C99 (excluding function declarations), and by adding support for variants,
 * sequences and CTF-specific specifiers. Enumeration container types
 * semantic is inspired from c++1x enum-base.
 */
```

C.1 Lexical grammar

C.1.1 LEXICAL ELEMENTS

```
token:
    keyword
    identifier
    constant
    string-literal
    punctuator
```

C.1.2 KEYWORDS

```
keyword: is one of

align
callsite
const
char
clock
double
enum
env
event
floating_point
float
integer
int
long
short
signed
stream
```

```
string
struct
trace
typealias
typedef
unsigned
variant
void
_Bool
_Complex
_Imaginary
```

C.1.3 **IDENTIFIERS**

```
identifier:
    identifier-nondigit
    identifier identifier-nondigit
    identifier digit

identifier-nondigit:
    nondigit
    universal-character-name
    any other implementation-defined characters

nondigit:
    _
    [a-zA-Z]    /* regular expression */

digit:
    [0-9]        /* regular expression */
```

C.1.4 **UNIVERSAL CHARACTER NAMES**

```
universal-character-name:
    \u hex-quad
    \U hex-quad hex-quad

hex-quad:
    hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit
```

C.1.5 **Constants**

```
constant:
    integer-constant
    enumeration-constant
    character-constant

integer-constant:
    decimal-constant integer-suffix-opt
    octal-constant integer-suffix-opt
    hexadecimal-constant integer-suffix-opt

decimal-constant:
    nonzero-digit
    decimal-constant digit

octal-constant:
    0
    octal-constant octal-digit

hexadecimal-constant:
    hexadecimal-prefix hexadecimal-digit
    hexadecimal-constant hexadecimal-digit

hexadecimal-prefix:
    0x
    0X

nonzero-digit:
    [1-9]

integer-suffix:
    unsigned-suffix long-suffix-opt
    unsigned-suffix long-long-suffix
    long-suffix unsigned-suffix-opt
    long-long-suffix unsigned-suffix-opt

unsigned-suffix:
    u
    U

long-suffix:
    l
    L

long-long-suffix:
    ll
    LL

enumeration-constant:
    identifier
    string-literal

character-constant:
    ' c-char-sequence '
    L' c-char-sequence '

c-char-sequence:
    c-char
    c-char-sequence c-char

c-char:
    any member of source charset except single-quote ('), backslash
    (\), or new-line character.
    escape-sequence

escape-sequence:
    simple-escape-sequence
    octal-escape-sequence
    hexadecimal-escape-sequence
    universal-character-name

simple-escape-sequence: one of
    \' \" \? \\ \a \b \f \n \r \t \v

octal-escape-sequence:
    \ octal-digit
    \ octal-digit octal-digit
    \ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:
    \x hexadecimal-digit
    hexadecimal-escape-sequence hexadecimal-digit
```

C.1.6 **STRING LITERALS**

```
string-literal:
    " s-char-sequence-opt "
    L" s-char-sequence-opt "

s-char-sequence:
    s-char
    s-char-sequence s-char

s-char:
    any member of source charset except double-quote ("), backslash
    (\), or new-line character.
    escape-sequence
```

C.1.7 **PUNCTUATORS**

```
punctuator: one of
    [ ] ( ) { } . -> * + - < > : ; ... = ,
```

C.2 **Phrase structure grammar**

```
primary-expression:
    identifier
    constant
    string-literal
    ( unary-expression )

postfix-expression:
    primary-expression
    postfix-expression [ unary-expression ]
    postfix-expression . identifier
    postfix-expressoin -> identifier

unary-expression:
    postfix-expression
    unary-operator postfix-expression

unary-operator: one of
    + -

assignment-operator:
    =

type-assignment-operator:
    :=

constant-expression-range:
    unary-expression ... unary-expression
```

C.2.2 **DECLARATIONS:**

```
declaration:
    declaration-specifiers declarator-list-opt ;
    ctf-specifier ;

declaration-specifiers:
    storage-class-specifier declaration-specifiers-opt
    type-specifier declaration-specifiers-opt
    type-qualifier declaration-specifiers-opt

declarator-list:
    declarator
    declarator-list , declarator

abstract-declarator-list:
    abstract-declarator
    abstract-declarator-list , abstract-declarator

storage-class-specifier:
    typedef

type-specifier:
    void
    char
    short
    int
    long
    float
    double
    signed
    unsigned
    _Bool
    _Complex
    _Imaginary
    struct-specifier
    variant-specifier
    enum-specifier
    typedef-name
    ctf-type-specifier

align-attribute:
    align ( unary-expression )

struct-specifier:
    struct identifier-opt { struct-or-variant-declaration-list-opt } align-attribute-opt
    struct identifier align-attribute-opt

struct-or-variant-declaration-list:
    struct-or-variant-declaration
    struct-or-variant-declaration-list struct-or-variant-declaration

struct-or-variant-declaration:
    specifier-qualifier-list struct-or-variant-declarator-list ;
    declaration-specifiers-opt storage-class-specifier declaration-specifiers-opt declarator-list ;
    typealias declaration-specifiers abstract-declarator-list type-assignment-operator declaration-specifiers a
    typealias declaration-specifiers abstract-declarator-list type-assignment-operator declarator-list ;

specifier-qualifier-list:
    type-specifier specifier-qualifier-list-opt
    type-qualifier specifier-qualifier-list-opt

struct-or-variant-declarator-list:
    struct-or-variant-declarator
    struct-or-variant-declarator-list , struct-or-variant-declarator

struct-or-variant-declarator:
    declarator
    declarator-opt : unary-expression

variant-specifier:
    variant identifier-opt variant-tag-opt { struct-or-variant-declaration-list }
    variant identifier variant-tag

variant-tag:
    < unary-expression >

enum-specifier:
    enum identifier-opt { enumerator-list }
    enum identifier-opt { enumerator-list , }
    enum identifier
    enum identifier-opt : declaration-specifiers { enumerator-list }
    enum identifier-opt : declaration-specifiers { enumerator-list , }

enumerator-list:
    enumerator
    enumerator-list , enumerator

enumerator:
    enumeration-constant
    enumeration-constant assignment-operator unary-expression
    enumeration-constant assignment-operator constant-expression-range

type-qualifier:
    const

declarator:
    pointer-opt direct-declarator

direct-declarator:
    identifier
    ( declarator )
    direct-declarator [ unary-expression ]

abstract-declarator:
    pointer-opt direct-abstract-declarator

direct-abstract-declarator:
    identifier-opt
    ( abstract-declarator )
    direct-abstract-declarator [ unary-expression ]
    direct-abstract-declarator [ ]
```

```
pointer:
    * type-qualifier-list-opt
    * type-qualifier-list-opt pointer

type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier

typedef-name:
    identifier
```

C.2.3 **CTF-SPECIFIC DECLARATIONS**

```
ctf-specifier:
    clock { ctf-assignment-expression-list-opt }
    event { ctf-assignment-expression-list-opt }
    stream { ctf-assignment-expression-list-opt }
    env { ctf-assignment-expression-list-opt }
    trace { ctf-assignment-expression-list-opt }
    callsite { ctf-assignment-expression-list-opt }
    typealias declaration-specifiers abstract-declarator-list type-assignment-operator declaration-specifiers a
    typealias declaration-specifiers abstract-declarator-list type-assignment-operator declarator-list

ctf-type-specifier:
    floating_point { ctf-assignment-expression-list-opt }
    integer { ctf-assignment-expression-list-opt }
    string { ctf-assignment-expression-list-opt }
    string

ctf-assignment-expression-list:
    ctf-assignment-expression ;
    ctf-assignment-expression-list ctf-assignment-expression ;

ctf-assignment-expression:
    unary-expression assignment-operator unary-expression
    unary-expression type-assignment-operator type-specifier
    declaration-specifiers-opt storage-class-specifier declaration-specifiers-opt declarator-list
    typealias declaration-specifiers abstract-declarator-list type-assignment-operator declaration-specifiers a
    typealias declaration-specifiers abstract-declarator-list type-assignment-operator declarator-list
```

EXAMPLES

This section contains pratical examples of CTF concepts. Each example shows a TSDL snippet, some binary data matched by this metadata, and the equivalent human-readable values (its format is *loosely based* on YAML).

For all the examples below, if the trace's native byte order (`byte_order` property of `trace` block; see 4.1.3 Byte order) is not specified, little-endian (`le`) is assumed.

Padding and *don't care* bits/bytes are indicated with black `x` s. The writer and the reader should not care about the values of those bits/bytes. A double `xx` in hexadecimal represents one byte of padding/*don't care*. A single `x` in binary represents one bit of padding/*don't care*.

Move your mouse over binary data or human-readable values sections to highlight the related TSDL parts. Click on sections to mark them.

Contents:

Types

> **SPECIFICATION SECTION:** 4. Types

The examples of this section demonstrate CTF types, and how their TSDL properties affect the way binary data is parsed.

Basic types

> **SPECIFICATION SECTION:** 4.1 Basic types

The basic CTF types are integers, floating point numbers, and enumerations.

INTEGERS

> **SPECIFICATION SECTION:** 4.1.5 Integers

A CTF integer is described by a TSDL `integer` block. Its only required property `size`, its size in bits, which has no default value. When `byte_order` is not specified, it defaults to `native` (little-endian for the sake of the following examples).

```
r {
ze = 16;
```

STREAM DATA (HEX)

```
0: 52 8f
```

VALUES

```
36690
```

CTF integers are unsigned by default. Use `signed = true` to make them signed:

```
r {
ze = 32;
gned = true;
te_order = be;
```

STREAM DATA (HEX)

```
0: fe d7 33 ea
```

VALUES

```
-19450902
```

CTF integers may have any size, including sizes which are not powers of two:

```
r {
ze = 23;
gned = true;
te_order = be;
```

STREAM DATA (BINARY)

```
0: 11011011 00100101 0110010X
```

VALUES

```
-1207630
```

When the byte order is little-endian, the remaining bits are always placed in the lower bits part of the byte:

```
r {
ze = 23;
gned = true;
te_order = le;
```

STREAM DATA (BINARY)

```
0: 10110010 10010010 X1101101
```

VALUES

```
-1207630
```

The `align` property of CTF integers is covered in structure examples, as it is only relevant when an integer is a structure field. Consecutive integers are also discussed in the structure examples.

FLOATING POINT NUMBERS

> SPECIFICATION SECTION: 4.1.7 Floating point

A floating point number can be written to a CTF stream. It is described using a `floating_point` block in TSDL. The mandatory properties of a `floating_point` block are `exp_dit` and `mant_dig`, which specify the number of bits respectively taken by the exponent part and by the mantissa part of the floating point number.

The following example shows a typical IEEE 754-2008 binary32 (single precision) floating point number:

STREAM DATA (HEX)

VALUES

```
ng_point {
p_dig = 8;
nt_dig = 24;
te_order = be;
```

```
0: c0 49 0f db
```

```
-3.1415927
```

The `byte_order` property of `floating_point` specifies the byte order:

```
ng_point {
p_dig = 8;
nt_dig = 24;
te_order = le;
```

STREAM DATA (HEX)

```
0: db 0f 49 c0
```

VALUES

```
-3.1415927
```

The `align` property of CTF floating point numbers is covered in structure examples, as it is only relevant when a floating point number is structure field.

ENUMERATIONS

SPECIFICATION SECTION: 4.1.8 Enumerations

CTF enumerations are mappings of ranges of integers to labels. The binary stream only contains an integer, and CTF readers use the metadata stream to display its equivalent label. The underlying integer type is specified after the `:` token. A type alias may also be used.

Here's a simple example, with single values mapped to labels:

```
 integer { size = 8; } {
NANA,     /* starts at 0 */
ANBERRY,  /* 1 */
NGERINE,  /* 2 */
G,        /* 3 */
```

STREAM DATA (HEX)

```
0: 02
```

VALUES

```
"TANGERINE" (2)
```

Explicit values can be mapped to labels by using `=`. Labels need to be surrounded by quotes (`"`) when they do not match the syntax of C identifiers:

```
 integer { size = 16; } {
NANA,          /* 0 */
ANBERRY,       /* 1 */
G,             /* 2 */
NGERINE = 6,
CONUT,         /* 7 */
LOOD ORANGE", /* 8 */
APE = 172,
MON,           /* 173 */
```

STREAM DATA (HEX)

```
0: 07 00
```

VALUES

```
"COCONUT" (7)
```

A range of integers can be mapped to one label by using the `...` operator:

```
 integer { size = 8; } {
NANA = 12,
ANBERRY,      /* 13 */
G = 17 ... 79,
PLE,          /* 80 */
NGERINE = 85,
LOOD ORANGE", /* 86 */
```

STREAM DATA (HEX)

```
0: 42
```

VALUES

```
"FIG" (42)
```

Compound types

SPECIFICATION SECTION: 4.2 Compound types

Compound CTF types are types including other CTF types. They are: arrays, sequences, strings, structures, and variants.

STRUCTURES

SPECIFICATION SECTION: 4.2.1 Structures

CTF structures are similar to C structures: they hold an ordered list of fields, each field having a name and a type. In CTF's case, field types are CTF types. They are represented in TSDL using a `struct` block.

Here's a simple CTF structure with three integer fields:

```
 {
teger {
  size = 16;
field1;

teger {
  size = 8;
  signed = true;
field2;

teger {
  size = 32;
  byte_order = be;
field3;
```

STREAM DATA (HEX)

```
0: 46 15 e9 01 32 8f 01
```

VALUES

```
field1: 5446
field2: -23
field3: 20090625
```

Fields are aligned within a structure (or within a compound type in general) following their implicit or explicit alignment. For example, a CTF integer is implicitly aligned on bits when its size is not a multiple of 8, and on bytes otherwise (8-bit alignment). The same goes for floating point numbers, considering their total sizes (`exp_dig + mant_dig`). Alignment can be made explicit with integers and floating point numbers by using the `align` property. Alignment values are always powers of two.

When concatenation of binary values is not possible due to alignment requirements, *extra padding* is inserted:

```
 {
 implicit 8-bit alignment */
teger {
  size = 16;
  byte_order = be;
field1;

 explicit 32-bit alignment */
oating_point {
  exp_dig = 8;
  mant_dig = 24;
  byte_order = be;
  align = 32;
field2;

 implicit 8-bit alignment */
teger {
  size = 8;
  signed = true;
field3;

 explicit 16-bit alignment */
teger {
  size = 8;
  align = 16;
field4;
```

STREAM DATA (HEX)

```
0: ab cd
2: XX XX
4: c0 49 0f db
8: d6
9: XX
a: fe
```

VALUES

```
field1: 43981
field2: -3.1415927
field3: -46
field4: 254
```

CTF structures may contain any CTF type as fields, including compound types:

```
 {
teger {
  size = 16;
field1;

ruct {
  integer {
      size = 8;
  } field1;

  integer {
      size = 32;
      signed = true;
  } field2;
field2;

oating_point {
  exp_dig = 8;
  mant_dig = 24;
  byte_order = be;
  align = 32;
field3;
```

STREAM DATA (HEX)

```
0: 39 30
2: aa 11 04 88 19
7: XX
8: 40 95 6a 16
```

VALUES

```
field1: 12345
field2:
  field1: 170
  field2: 428344337
field3: 4.6692
```

A CTF structure's alignment is equal to the maximum alignment amongst all the contained basic types. This ensures that inner basic types are always properly aligned:

```
 {
teger {
  size = 8;
field1;

 32-bit alignment */
ruct {
  /* 8-bit alignment */
  integer {
      size = 8;
  } field1;

  /* 32-bit alignment */
  integer {
      size = 8;
      align = 32;
  } field2;
field2;

teger {
  size = 8;
field3;
```

STREAM DATA (HEX)

```
0: 42
1: XX XX XX
4: 17
5: XX XX XX
8: b1 07 00 00
c: ff
```

VALUES

```
field1: 66
field2:
  field1: 23
  field2: 1969
field3: 255
```

As you can see, when fields are aligned on their own sizes, it's always preferable, for space optimization, to put them in descending order of size (larger fields first, then smaller ones). Here's the same example, albeit with bigger fields placed first (field names are unchanged). It is clear that six bytes of padding were saved by carefully ordering the fields:

```
 {
 32-bit alignment */
ruct {
  /* 32-bit alignment */
  integer {
      size = 8;
      align = 32;
  } field2;

  /* 8-bit alignment */
  integer {
      size = 8;
  } field1;
field2;

teger {
  size = 8;
field1;

teger {
  size = 8;
field3;
```

STREAM DATA (HEX)

```
0: b1 07 00 00
4: 17
5: 42
6: ff
```

VALUES

```
field2:
  field2: 1969
  field1: 23
field1: 66
field3: 255
```

If a CTF structure's alignment is required to be *larger than* the larger alignment amongst its contained basic types, the trailing `align()` attribute may be used:

```
 {
teger {
```

STREAM DATA (HEX)

```
0: 42
1: XX XX XX XX XX XX XX
```

VALUES

```
field1: 66
field2:
  size = 8;
field1;

 forced 64-bit alignment */
ruct {
  /* 32-bit alignment */
  integer {
      size = 8;
      align = 32;
  } field1;

  /* 8-bit alignment */
  integer {
      size = 8;
  } field2;
align(64) field2;

teger {
  size = 8;
field3;
```

```
8: b1 07 00 00
c: 17 ff
```

```
  field1: 1969
  field2: 23
field3: 255
```

ARRAYS

SPECIFICATION SECTION: 4.2.3 Arrays

A CTF array is a fixed-length list of consecutive CTF types. An array's length is defined in the metadata.

Creating a CTF array is done by using the `[]` operator after a field name:

```
 {
teger {
  size = 16;
simple_field;

teger {
  size = 8;
array_field[8];

teger {
  size = 8;
other_simple_field;
```

STREAM DATA (HEX)

```
0: 21 f8
2: 00 01 01 02 03 05 08 0d
a: 55
```

VALUES

```
simple_field: 63521
array_field:
  [0]: 0
  [1]: 1
  [2]: 1
  [3]: 2
  [4]: 3
  [5]: 5
  [6]: 8
  [7]: 13
other_simple_field: 85
```

Multidimensional arrays are allowed, borrowing the C language syntax. The following example describes an array of three pairs of bytes integers:

```
 {
teger {
  size = 16;
simple_field;

teger {
  size = 8;
multi_array_field[3][2];

teger {
  size = 8;
other_simple_field;
```

STREAM DATA (HEX)

```
0: 21 f8
2: 00 01 01 02 03 05
8: 55
```

VALUES

```
simple_field: 63521
multi_array_field:
  [0][0]: 0
  [0][1]: 1
  [1][0]: 1
  [1][1]: 2
  [2][0]: 3
  [2][1]: 5
other_simple_field: 85
```

The alignment of an array's underlying type must be respected when repeating elements:

```
 {
teger {
  size = 16;
simple_field;

teger {
  size = 8;
  align = 16;
array_field[5];

teger {
  size = 8;
other_simple_field;
```

STREAM DATA (HEX)

```
0: 21 f8
2: 00
3: XX
4: 01
5: XX
6: 01
7: XX
8: 02
9: XX
a: 03 55
```

VALUES

```
simple_field: 63521
array_field:
  [0]: 0
  [1]: 1
  [2]: 1
  [3]: 2
  [4]: 3
other_simple_field: 85
```

Any CTF type may be repeated in an array:

```
 {
teger {
  size = 16;
simple_field;

ruct {
  integer {
      size = 8;
  } x;

  integer {
      size = 8;
  } y;
array_field[5];

teger {
  size = 8;
other_simple_field;
```

STREAM DATA (HEX)

```
0: 21 f8
2: 17 37 b1 2a fe 01
8: 65 c9 06 07 55
```

VALUES

```
simple_field: 63521
array_field:
  [0]: x: 23
       y: 55
  [1]: x: 177
       y: 42
  [2]: x: 254
       y: 1
  [3]: x: 101
       y: 201
  [4]: x: 6
       y: 7
other_simple_field: 85
```

SEQUENCES

> **SPECIFICATION SECTION:** 4.2.4 Sequences

CTF sequences are just like CTF arrays, except that their actual length is not statically defined by the metadata, but rather known dynamically, by looking at the current value of a previous CTF integer field.

Sequences share the arrays' TSDL syntax, with the exception that the `[]` operator contains an expression instead of a constant integer. The latter expression refers to a previous field. The following examples always use a field contained in the same structure to define a sequence's length. You should know, however, that it is possible to use the fields of other scopes (see stream packet examples structure).

Here's a sequence of bytes. Its length is determined by the dynamic value of the preceding `len` field:

```
 {
teger {
  size = 16;
len;

oating_point {
  exp_dig = 8;
  mant_dig = 24;
  byte_order = be;
some_float;

teger {
  size = 8;
my_sequence[len];
```

STREAM DATA (HEX)

```
0: 07 00 c0 49 0f db
6: 3d 4c 2f 05 58 17 34
```

VALUES

```
len: 7
some_float: -3.1415927
my_sequence:
  [0]: 61
  [1]: 76
  [2]: 47
  [3]: 05
  [4]: 88
  [5]: 23
  [6]: 52
```

Multidimensional sequences are allowed, possibly mixed with arrays:

```
 {
teger {
  size = 8;
len2;

teger {
  size = 8;
len1;

ruct {
  integer {
      size = 8;
  } a;

  integer {
      size = 8;
  } b;
align(32) seq[len1][len2];

teger {
  size = 16;
  align = 64;
famous_last_int;
```

STREAM DATA (HEX)

```
00: 02 03
02: XX XX
04: 01 02
06: XX XX
08: 03 04
0a: XX XX
0c: 0a 0b
0e: XX XX
10: 0c 0d
12: XX XX
14: ff fe
16: XX XX
18: fd fc
1a: XX XX XX XX XX XX
20: 42 42
```

VALUES

```
len2: 2
len1: 3
seq:
  [0][0]: a: 1
          b: 2
  [0][1]: a: 3
          b: 4
  [1][0]: a: 10
          b: 11
  [1][1]: a: 12
          b: 13
  [2][0]: a: 255
          b: 254
  [2][1]: a: 253
          b: 252
famous_last_int: 16962
```

STRINGS

> **SPECIFICATION SECTION:** 4.2.5 Strings

CTF strings are arrays of bytes (8-bit integers) terminated by the null character (byte with the value 0). The default CTF string encoding is UTF-8.

```
 {
teger {
  size = 16;
```

STREAM DATA (HEX)

```
00: 23 62
02: 49 20 3c 33 20 43 54 46 00
```

VALUES

```
some_int: 25123
my_string: "I <3 CTF"
other_int: 1729
```

```
some_int;

ring my_string;

teger {
  size = 32;
  align = 32;
other_int;
```

```
0b: XX
0c: c1 06 00 00
```

VARIANTS

> **SPECIFICATION SECTION:** 4.2.2 Variants (discriminated/tagged unions)

CTF variants allow any CTF type to be selected dynamically using a previously defined CTF enumeration. They are similar to C unions, although a C union's selection is known at build time, whereas a CTF variant's selection is known at run time.

The CTF enumeration used for dynamically selecting a variant's current type is called the variant's *tag*. Each field of a variant has a unique name within its scope. The tag's current label (determined by the enumeration's current value) determines which field of the variant is selected.

The following example shows a CTF variant with three options:

- an 8-bit integer
- a string
- a 32-bit floating point number

The variant's tag is previous field `my_tag`:

```
 {
 variant's tag */
um : integer { size = 8; } {
  INT, STRING, FLOAT,
my_tag;

riant <my_tag> {
  integer {
      size = 8;
  } INT;

  string STRING;

  floating_point {
      exp_dig = 8;
      mant_dig = 24;
      byte_order = be;
  } FLOAT;
my_variant;
```

STREAM DATA (HEX)

```
0: 02 c0 49 0f db
```

VALUES

```
my_tag: "FLOAT" (2)
my_variant: -3.1415927
```

The effective alignment of a CTF variant type is the alignment of its current selected field, *not* the maximum alignment amongst its possible choices:

STREAM DATA (HEX)

VALUES

```
 {
 variant's tag */
um : integer { size = 8; } {
  STRING, INT, FLOAT,
my_tag;

 UTF-8 string */
ring str;

riant <my_tag> {
  /* 8-bit alignment */
  string STRING;

  /* 16-bit alignment */
  integer {
      size = 16;
      align = 16;
  } INT;

  /* 32-bit alignment */
  floating_point {
      exp_dig = 8;
      mant_dig = 24;
      byte_order = be;
      align = 32;
  } FLOAT;
my_variant;
```

```
0: 01
1: 4d 6f 6e 74 72 c3 a9 61 6c 00
b: XX
c: 15 23
```

```
my_tag: "INT" (1)
str: "Montréal"
my_variant: 8981
```

Type aliases and named types

Now that you are familiar with CTF types, you should know that it is possible to create aliases for them, somehow like the C language allows with `typedef`. CTF's equivalent is `typealias`:

```
ias integer {
ze = 8;
int8_t;

 {
nt8_t field1;
nt8_t field2;
```

STREAM DATA (HEX)

```
0: 23 42
```

VALUES

```
field1: 35
field1: 66
```

C types and type qualifiers may be used as CTF type alias names:

```
ias integer {
ze = 8;
onst unsigned char;

 {
nst unsigned char field1;
nst unsigned char field2;
```

STREAM DATA (HEX)

```
0: 23 42
```

VALUES

```
field1: 35
field1: 66
```

Here's another example, aliasing a CTF structure with a forced 32-bit alignment:

STREAM DATA (HEX)

VALUES

```
ias struct {
teger {
  size = 16;
  signed = true;
a;

teger {
  size = 8;
b;
n(32) := my_struct;

 {
_struct field1;
_struct field2;
```

```
0: 01 ab 58
3: XX
4: dc ff 03
```

```
field1:
  a: -21759
  b: 88
field2:
  a: -36
  b: 3
```

It is also possible to declare named enumerations, structures, and variants in any lexical scope:

```
as for 8-bit integer */
ias integer {
ze = 8;
yte;

as for 32-bit float */
ias floating_point {
p_dig = 8;
nt_dig = 24;
te_order = be;
ign = 32;
loat;

ed enumeration */
y_enum : byte {
TE, FLOAT,


ed variant */
t my_variant {
te BYTE;
oat FLOAT;


ed structure */
 my_struct {
um my_enum tag;
te some_byte;

 using a named variant */
riant my_variant <some_byte> var;


 {
te this_byte;
ruct my_struct this_struct;
```

STREAM DATA (HEX)

```
0: 23 01 fe
3: XX
4: 40 2d f8 54
```

VALUES

```
this_byte: 35
this_struct:
  tag: "FLOAT" (1)
  some_byte: 254
  var: 2.7182817
```

Notice how type keywords are placed before the names in this case (e.g., `struct my_struct` and `variant my_variant`), whereas only the alias needs to be specified when referring to type aliases.

Stream packet structure

> **SPECIFICATION SECTIONS:** 5. Event packet header and 6. Event structure

The examples of this section show complete CTF streams. Unless otherwise mentioned, assume all the examples of this section start with the following block:

```
/* CTF 1.8 */

typealias integer {size = 8;}  := uint8_t;
typealias integer {size = 16;} := uint16_t;
typealias integer {size = 32;} := uint32_t;
typealias integer {size = 8; signed = true;}  := int8_t;
typealias integer {size = 16; signed = true;} := int16_t;
typealias integer {size = 32; signed = true;} := int32_t;

typealias floating_point {
    exp_dig = 8;
    mant_dig = 24;
    align = 32;
} := float;

typealias floating_point {
    exp_dig = 11;
    mant_dig = 53;
    align = 64;
} := double;
```

Minimal examples

The following example shows the bare minimum CTF trace. It has no packet header, which means it has a single packet. It has no `clock` block, as clocks are optional. It has no `stream` block, which means it has a single stream. It has only one `event` block, which only needs a name (empty in this case), and automatically belongs to the single stream. The event has a single byte as its payload (it could also be a single bit, but a byte simplifies the example below). The metadata of this example does *not* start with the common block above, i.e. it is complete. The binary stream contains three events:

```
 1.8 */

{
jor = 1;
nor = 8;
te_order = le;


{
me = "";
elds := struct {
  integer {
      size = 8;
  } a_byte;
```

STREAM DATA (BINARY)

```
0: ab cd ef
```

VALUES

```
packets:
  - events:
    - name: ""
      fields:
        a_byte: 171
    - name: ""
      fields:
        a_byte: 205
    - name: ""
      fields:
        a_byte: 239
```

Of course, this example is too simple to be anything useful for tracing purposes. Let us add a packet header, a clock, a stream, and use a structure as the type of `event.fields`. Since this stream has no packet context to indicate its packet and content sizes, it is considered to hold a single packet. The stream of this example also contains three events.

```
{
jor = 1;
nor = 8;
te_order = le;
cket.header := struct {
  uint32_t magic;
  uint32_t stream_id;


{
me = my_clock;
eq = 1000;
fset_s = 1421703448;


ias integer {
ze = 32;
p = clock.my_clock.value;
y_clock_int_t;

 {
 = 0;
ent.header := struct {
  uint32_t id;
  my_clock_int_t timestamp;



{
 = 0;
me = "my_event";
ream_id = 0;
elds := struct {
  uint32_t a;
  uint16_t b;
  string c;
```

STREAM DATA (HEX)

```
00: c1 1f fc c1 00 00 00 00
08: 00 00 00 00 90 47 05 00
10: 78 56 34 12 cd ab
16: 6a 73 6d 69 74 68 00
1d: 00 00 00 00 3c 3d 09 00
25: 00 ef cd ab 42 42
2b: 62 61 63 6f 6e 00
31: 00 00 00 00 62 06 1d 00
39: aa 55 aa 55 34 00
3f: 4c 69 6e 75 78 00
```

VALUES

```
packets:
  - header:
      magic: 0xc1fc1fc1
      stream_id: 0
    events:
      - name: "my_event"
        header:
          id: 0
          timestamp: 346000
        fields:
          a: 305419896
          b: 43981
          c: "jsmith"
      - name: "my_event"
        header:
          id: 0
          timestamp: 605500
        fields:
          a: 2882400000
          b: 16962
          c: "bacon"
      - name: "my_event"
        header:
          id: 0
          timestamp: 1902178
        fields:
          a: 1437226410
          b: 52
          c: "Linux"
```

Stream packet context

A packet context may be specified at the stream level, which means different streams may have different packet context layouts. All the packets of a given stream will share the same packet context type, though.

The stream packet context immediately follows the packet header, without considering padding. It's usually a structure, potentially containing the following fields:

- `packet_size` : the total size of the packet in bits
- `content_size` : the total size of the packet's content in bits
- `timestamp_begin` : the packet creation timestamp
- `timestamp_end` : the packet flushing timestamp

The packet context may contain anything related to the packet itself. It is also common to put the processor ID in there, provided that a given stream is always written to by a single processor.

Here's our minimal CTF trace with added packet context:

STREAM DATA (HEX)

VALUES

```
{
jor = 1;
nor = 8;
te_order = le;
cket.header := struct {
  uint32_t magic;
  uint32_t stream_id;



{
me = my_clock;
eq = 1000;
fset_s = 1421703448;


ias integer {
ze = 32;
p = clock.my_clock.value;
y_clock_int_t;

 {
 = 0;
cket.context := struct {
  uint32_t packet_size;
  uint32_t content_size;
  my_clock_int_t timestamp_begin;
  my_clock_int_t timestamp_end;
  int16_t something_else;
  uint8_t cpu_id;

ent.header := struct {
  uint32_t id;
  my_clock_int_t timestamp;



{
 = 0;
me = "my_event";
ream_id = 0;
elds := struct {
  uint32_t a;
  uint16_t b;
  string c;
```

```
00: c1 1f fc c1 00 00 00 00
08: 30 03 00 00 c0 02 00 00
10: 01 18 00 00 04 2c 1d 00
18: 10 ab 02
1b: 00 00 00 00 90 47 05 00
23: 78 56 34 12 cd ab
29: 6a 73 6d 69 74 68 00
30: 00 00 00 00 3c 3d 09 00
38: 00 ef cd ab 42 42
3e: 62 61 63 6f 6e 00
44: 00 00 00 00 62 06 1d 00
4c: aa 55 aa 55 34 00
52: 4c 69 6e 75 78 00
58: XX XX XX XX XX XX XX XX
60: XX XX XX XX XX XX
```

```
packets:
  - header:
      magic: 0xc1fc1fc1
      stream_id: 0
    context:
      packet_size: 816
      content_size: 704
      timestamp_begin: 6145
      timestamp_end: 1911812
      something_else: -21744
      cpu_id: 2
    events:
      - name: "my_event"
        header:
          id: 0
          timestamp: 346000
        fields:
          a: 305419896
          b: 43981
          c: "jsmith"
      - name: "my_event"
        header:
          id: 0
          timestamp: 605500
        fields:
          a: 2882400000
          b: 16962
          c: "bacon"
      - name: "my_event"
        header:
          id: 0
          timestamp: 1902178
        fields:
          a: 1437226410
          b: 52
          c: "Linux"
```

Multiple streams

Here's an example showing multiple streams. The first stream, stream 0, has two events, named `my_event` and `my_other_event` . The second stream, stream 1, has one event, named `yet_another` . Both stream use the same structure (`struct ev_header`) as their event header.

Event IDs and names must be unique per stream. In this example, two events have the same ID (0), but belong to different streams, thus it is allowed.

```
{
jor = 1;
nor = 8;
te_order = le;
cket.header := struct {
  uint32_t magic;
  uint32_t stream_id;



{
me = my_clock;
```

STREAM DATA (HEX)

Stream 0:

```
00: c1 1f fc c1 00 00 00 00
08: 18 02 00 00 f8 01 00 00
10: 00
11: 00 00 00 00 90 47 05 00
19: 2f 74 6d 70 00
1e: 01 00 00 00 ff 01 13 00
26: XX XX
28: 19 4e 76 cc 11 22 33 44
30: 00 00 00 00 08 cd 2f 00
38: 68 75 6d 6d 75 73 00
3f: XX XX XX XX
```

VALUES

Stream 0:

```
  packets:
    - header:
        magic: 0xc1fc1fc1
        stream_id: 0
      context:
        packet_size: 536
        content_size: 504
        cpu_id: 0
      events:
        - name: "my_event"
          header:
```

```
eq = 1000;
fset_s = 1421703448;


ias integer {
ze = 32;
p = clock.my_clock.value;
y_clock_int_t;

 ev_header {
nt32_t id;
_clock_int_t timestamp;


 {
 = 0;
ent.header := struct ev_header;
cket.context := struct {
  uint32_t packet_size;
  uint32_t content_size;
  uint8_t cpu_id;



 {
 = 1;
ent.header := struct ev_header;


{
 = 0;
me = "my_event";
ream_id = 0;
elds := struct {
  string a;



{
 = 1;
me = "my_other_event";
ream_id = 0;
elds := struct {
  uint32_t a;
  uint32_t b;
align(64);


{
 = 0;
me = "yet_another";
ream_id = 1;
elds := struct {
  uint32_t len;
  string strings[len];
```

```
Stream 1:
00: c1 1f fc c1 01 00 00 00
08: 00 00 00 00 12 34 56 00
10: 03 00 00 00
14: 6d 65 6f 77 00
19: 74 72 61 63 69 6e 67 00
21: 77 61 76 65 73 00
08: 00 00 00 00 ab cd ef 00
10: 02 00 00 00
14: 73 68 61 6d 72 6f 63 6b 00
19: 47 75 69 7a 6f 74 00
```

```
          id: 0
          timestamp: 346000
        fields:
          a: "/tmp"
      - name: "my_other_event"
        header:
          id: 1
          timestamp: 1245695
        fields:
          a: 3430305305
          b: 1144201745
      - name: "my_event"
        header:
          id: 0
          timestamp: 3132680
        fields:
          a: "hummus"

Stream 1:
  packets:
    - header:
        magic: 0xc1fc1fc1
        stream_id: 1
      events:
      - name: "yet_another"
        header:
          id: 0
          timestamp: 5649426
        fields:
          len: 3
          strings:
            [0]: "meow"
            [1]: "tracing"
            [2]: "waves"
      - name: "yet_another"
        header:
          id: 0
          timestamp: 15715755
        fields:
          len: 2
          strings:
            [0]: "shamrock"
            [1]: "Guizot"
```

Scopes

> **SPECIFICATION SECTION:** 7.3 TSDL scopes

TSDL scopes are an advanced, yet important concept of the language. TSDL uses three different types of scoping: a *lexical scope* is used for declarations and type definitions, and *static and dynamic scopes* are used for variant references to tag fields (with relative and absolute path lookups) as well as for sequence references to length fields.

This section shows practical examples of how the different types of scopes work.

Lexical scope

> **SPECIFICATION SECTION:** 7.3.1 Lexical scope

Lexical scopes are used to contain declarations and type definitions (named types and type aliases).

A TSDL lexical scope is created using the `{` symbol. Everything following belongs to the new nested lexical scope, which ends with its associated `}` symbol (at the same level).

An inner lexical scope can refer to types declared within its container lexical scope *prior to* the inner declaration lexical. Redefinition of a type definition or type alias is not valid, although hiding an upper scope type definition/alias is allowed within a sub-scope.

The `trace`, `env`, `stream`, `event`, `struct`, and `variant` blocks have their own lexical scope. A root lexical scope also contains all declarations located outside of any of the aforementioned scopes.

Here's an example: the following inner structure may use a type alias defined in its container (another structure), as long as this type alias is defined before it:

```
struct {
    typealias integer {
        size = 8;
    } := byte;

    /* may use byte, cannot use word */
    struct my_struct {
        byte a;
        byte b;
        byte c;
    };

    typealias integer {
        size = 16;
    } := word;
}
```

Here's an example of lexical scope shadowing. Both events `e0` and `e1` have the same payload (a unique field of type `my_int`). However, event `e0` has an inner type alias which shadows the upper scope one since they share the same name:

```
/* CTF 1.8 */

trace {
    major = 1;
    minor = 8;
    byte_order = le;
};

typealias integer {
    size = 8;
} := my_int;

stream {
    event.header := struct {
        my_int id;
    };
};

event {
    name = "e0";
    id = 0;

    typealias integer {
        size = 16;
    } := my_int;

    fields := struct {
        my_int field;
    };
};

event {
    name = "e1";
    id = 1;

    fields := struct {
        my_int field;
    };
};
```

Note that the very first type alias in the example above is defined in the *root scope*, i.e. the one directly containing the `trace`, `env`, `clock`, `stream`, and `event` blocks.

Static scope

> **SPECIFICATION SECTION:** 7.3.2 Static and dynamic scopes

Static scopes are used for variant tag and sequence length lookups. They include the immediate previous structure/variant fields, as well as any sub-scope of those.

The following example shows a few possible lookups within a static scope: previous field in same local static scope, previous field in parent static scope, and sub-scope of previous field using the dot notation:

```
 {
 alias for a byte */
pealias integer {
  size = 8;
:= byte;

te len;

ruct {
  byte len2;

  /* parent lookup */
  byte bytes[len];

  /* local lookup */
  byte bytes2[len2];
the_bytes;

 sub-scope lookup */
te bytes[the_bytes.len2];
```

STREAM DATA (HEX)

```
0: 03 04 ff fd fb
5: 03 12 19 87
9: 25 01 19 88
```

VALUES

```
len: 3
the_bytes:
  len2: 4
  bytes:
    [0]: 255
    [1]: 253
    [2]: 251
  bytes2:
    [0]: 3
    [1]: 18
    [2]: 25
    [3]: 135
bytes:
  [0]: 37
  [1]: 1
  [2]: 25
  [3]: 136
```

Dynamic scope

> **SPECIFICATION SECTION:** 7.3.2 Static and dynamic scopes

Static scopes have their limitation. For example, if you define a sequence/variant in the fields of an event, you cannot use a field of the event context, the stream packet context or the packet header, for example, as a length/tag, since the lookups are made by going up lexically, and the aforementioned scopes have no parent-child relationship.

Dynamic scoping is the ultimate feature of TSDL to select any previous field, in any scope, as sequence lengths and variant tags.

The most straightforward way of using dynamic scopes is writing absolute paths, prefixed with one of:

- Trace environment: `env.`
- Trace packet header: `trace.packet.header.`
- Stream packet context: `stream.packet.context.`
- Event header: `stream.event.header.`
- Stream event context: `stream.event.context.`
- Event context: `event.context.`
- Event payload: `event.fields.`

Here's an example showing a few of them:

```
{
jor = 1;
nor = 8;
te_order = le;
cket.header := struct {
  uint32_t magic;
  uint32_t stream_id;




n = 3;


{
me = my_clock;
eq = 1000;
fset_s = 1421703448;


ias integer {
ze = 32;
p = clock.my_clock.value;
y_clock_int_t;

 {
 = 0;
ent.header := struct {
  uint32_t id;
  my_clock_int_t timestamp;
  uint16_t length;




{
 = 0;
me = "my_event";
ream_id = 0;
ntext := struct {
  uint32_t a;
  uint8_t b[env.len];

elds := struct {
  uint32_t c;
  uint8_t d[event.context.a];
  string e[stream.event.
```

STREAM DATA (HEX)

```
00: c1 1f fc c1 00 00 00 00
08: 00 00 00 00 90 47 05 00
10: 03 00 02 00 00 00
16: ab cd ef 15 4e 64 ab
1d: 19 88
1f: 61 6c 64 65 72 00
25: 63 72 65 73 73 00
2b: 64 69 6e 64 6c 65 00
```

VALUES

```
packets:
  - header:
      magic: 0xc1fc1fc1
      stream_id: 0
    events:
      - name: "my_event"
        header:
          id: 0
          timestamp: 346000
          length: 3
        context:
          a: 2
          b:
            [0]: 171
            [1]: 205
            [2]: 239
        fields:
          c: 2875477525
          d:
            [0]: 25
            [1]: 136
          e:
            [0]: "alder"
            [1]: "cress"
            [2]: "dindle"
```

```
header.length];
```

Absolute paths allow any field to be reached in the current stream, current packet, and current event. Within an event, there exist an implicit priority between its scopes which may be used to avoid always using absolute paths (top one has the highest priority):

1. Event payload
2. Stream event context
3. Event context
4. Event header

That is, if you specify a variant tag/sequence length named `field`, and that `field` does not exist in the static scope, the event context will be examined, then the stream event context, and finally the event header. Here's an example:

```
{
jor = 1;
nor = 8;
te_order = le;
cket.header := struct {
  uint32_t magic;
  uint32_t stream_id;


{
me = my_clock;
eq = 1000;
fset_s = 1421703448;


ias integer {
ze = 32;
p = clock.my_clock.value;
y_clock_int_t;

 {
 = 0;
ent.header := struct {
  uint32_t id;
  my_clock_int_t timestamp;
  uint16_t length;



{
 = 0;
me = "my_event";
ream_id = 0;
ntext := struct {
  uint32_t len;
  uint8_t bytes[length];

elds := struct {
  uint8_t bytes[len];
  uint8_t bytes2[length];
```

STREAM DATA (HEX)

```
00: c1 1f fc c1 00 00 00 00
08: 00 00 00 00 90 47 05 00
10: 03 00 05 00 00 00
16: cd ab ff 01 02 03 04 05
1e: 40 50 60
```

VALUES

```
packets:
  - header:
      magic: 0xc1fc1fc1
      stream_id: 0
    events:
      - name: "my_event"
        header:
          id: 0
          timestamp: 346000
          length: 3
        context:
          len: 5
          bytes:
            [0]: 205
            [1]: 171
            [2]: 255
        fields:
          bytes:
            [0]: 1
            [1]: 2
            [2]: 3
            [3]: 4
            [4]: 5
          bytes2:
            [0]: 64
            [1]: 80
            [2]: 96
```