The CTF File Format

Version 3

Nick Alcock

Table of Contents

Overview

The CTF file format compactly describes C types and the association between function and data symbols and types: if embedded in ELF objects, it can exploit the ELF string table to reduce duplication further. There is no real concept of namespacing: only top-level types are described, not types scoped to within single functions.

CTF dictionaries can be *children* of other dictionaries, in a one-level hierarchy: child dictionaries can refer to types in the parent, but the opposite is not sensible (since if you refer to a child type in the parent, the actual type you cited would vary depending on what child was attached). This parent/child definition is recorded in the child, but only as a recommendation: users of the API have to attach parents to children explicitly, and can choose to attach a child to any parent they like, or to none, though doing so might lead to unpleasant consequences like dangling references to types. See Section 2.3.2 [Type indexes and type IDs], page 9. Type lookups in child dicts that are not associated with a parent at all will fail with `ECTF_NOPARENT` if a parent type was needed.

The associated API to generate, merge together, and query this file format will be described in the accompanying `libctf` manual once it is written. There is no API to modify dictionaries once they've been written out: CTF is a write-once file format. (However, it is always possible to dynamically create a new child dictionary on the fly and attach it to a pre-existing, read-only parent.)

There are two major pieces to CTF: the *archive* and the *dictionary*. Some relatives and ancestors of CTF call dictionaries *containers*: the archive format is unique to this variant of CTF. (Much of the source code still uses the old term.)

The archive file format is a very simple mmappable archive used to group multiple dictionaries together into groups: it is expected to slowly go away and be replaced by other mechanisms, but right now it is an important part of the file format, used to group dictionaries containing types with conflicting definitions in different TUs with the overarching dictionary used to store all other types. (Even when archives go away, the `libctf` API used to access them will remain, and access the other mechanisms that replace it instead.)

The CTF dictionary consists of a *preamble*, which does not vary between versions of the CTF file format, and a *header* and some number of *sections*, which can vary between versions.

The rest of this specification describes the format of these sections, first for the latest version of CTF, then for all earlier versions supported by `libctf`: the earlier versions are defined in terms of their differences from the next later one. We describe each part of the format first by reproducing the C structure which defines that part, then describing it at greater length in terms of file offsets.

The description of the file format ends with a description of relevant limits that apply to it. These limits can vary between file format versions.

This document is quite young, so for now the C code in '`ctf.h`' should be presumed correct when this document conflicts with it.

1 CTF archives

The CTF archive format maps names to CTF dictionaries. The names may contain any character other than \0, but for now archives containing slashes in the names may not extract correctly. It is possible to insert multiple members with the same name, but these are quite hard to access reliably (you have to iterate through all the members rather than opening by name) so this is not recommended.

CTF archives are not themselves compressed: the constituent components, CTF dictionaries, can be compressed. (See Section 2.2 [CTF header], page 5).

CTF archives usually contain a collection of related dictionaries, one parent and many children of that parent. CTF archives can have a member with a *default name*, `.ctf` (which can be represented as `NULL` in the API). If present, this member is usually the parent of all the children, but it is possible for CTF producers to emit parents with different names if they wish (usually for backward- compatibility purposes).

`.ctf` sections in ELF objects consist of a single CTF dictionary rather than an archive of dictionaries if and only if the section contains no types with identical names but conflicting definitions: if two conflicting definitions exist, the deduplicator will place the type most commonly referred to by other types in the parent and will place the other type in a child named after the translation unit it is found in, and will emit a CTF archive containing both dictionaries instead of a raw dictionary. All types that refer to such conflicting types are also placed in the per-translation-unit child.

The definition of an archive in '`ctf.h`' is as follows:

```
struct ctf_archive
{
  uint64_t ctfa_magic;
  uint64_t ctfa_model;
  uint64_t ctfa_nfiles;
  uint64_t ctfa_names;
  uint64_t ctfa_ctfs;
};

typedef struct ctf_archive_modent
{
  uint64_t name_offset;
  uint64_t ctf_offset;
} ctf_archive_modent_t;
```

(Note one irregularity here: the `ctf_archive_t` is not a typedef to `struct ctf_archive`, but a different typedef, private to `libctf`, so that things that are not really archives can be made to appear as if they were.)

All the above items are always in little-endian byte order, regardless of the machine endianness.

The archive header has the following fields:

Offset	Name	Description

0x00	`uint64_t ctfa_magic`	The magic number for archives, `CTFA_MAGIC`: 0x8b47f2a4d7623eeb.
0x08	`uint64_t ctfa_model`	The data model for this archive: an arbitrary integer that serves no purpose but to be handed back by the libctf API. See Section 2.8 [Data models], page 21.
0x10	`uint64_t ctfa_nfiles`	The number of CTF dictionaries in this archive.
0x18	`uint64_t ctfa_names`	Offset of the name table, in bytes from the start of the archive. The name table is an array of `struct ctf_archive_modent_t[ctfa_nfiles]`.
0x20	`uint64_t ctfa_ctfs`	Offset of the CTF table. Each element starts with a `uint64_t` size, followed by a CTF dictionary.

The array pointed to by `ctfa_names` is an array of entries of `ctf_archive_modent`:

Offset	**Name**	**Description**
0x00	`uint64_t name_offset`	Offset of this name, in bytes from the start of the archive.
0x08	`uint64_t ctf_offset`	Offset of this CTF dictionary, in bytes from the start of the archive.

The `ctfa_names` array is sorted into ASCIIbetical order by name (i.e. by the result of dereferencing the `name_offset`).

The archive file also contains a name table and a table of CTF dictionaries: these are pointed to by the structures above. The name table is a simple strtab which is not required to be sorted; the dictionary array is described above in the entry for `ctfa_ctfs`.

The relative order of these various parts is not defined, except that the header naturally always comes first.

2 CTF dictionaries

CTF dictionaries consist of a header, starting with a premable, and a number of sections.

2.1 CTF Preamble

The preamble is the only part of the CTF dictionary whose format cannot vary between versions. It is never compressed. It is correspondingly simple:

```
typedef struct ctf_preamble
{
  unsigned short ctp_magic;
  unsigned char ctp_version;
  unsigned char ctp_flags;
} ctf_preamble_t;
```

`#define`s are provided under the names `cth_magic`, `cth_version` and `cth_flags` to make the fields of the `ctf_preamble_t` appear to be part of the `ctf_header_t`, so consuming programs rarely need to consider the existence of the preamble as a separate structure.

Offset	Name	Description
0x00	`unsigned short ctp_magic`	The magic number for CTF dictionaries, `CTF_MAGIC`: 0xdff2.
0x02	`unsigned char ctp_version`	The version number of this CTF dictionary.
0x03	`ctp_flags`	Flags for this CTF file. See Section 2.1.1 [CTF file-wide flags], page 5.

Every element of a dictionary must be naturally aligned unless otherwise specified. (This restriction will be lifted in later versions.)

CTF dictionaries are stored in the native endianness of the system that generates them: the consumer (e.g., `libctf`) can detect whether to endian-flip a CTF dictionary by inspecting the `ctp_magic`. (If it appears as 0xf2df, endian-flipping is needed.)

The version of the CTF dictionary can be determined by inspecting `ctp_version`. The following versions are currently valid, and `libctf` can read all of them:

Version	Number	Description
`CTF_VERSION_1`	1	First version, rare. Very similar to Solaris CTF.
`CTF_VERSION_1_UPGRADED_3`	2	First version, upgraded to v3 or higher and written out again. Name may change. Very rare.
`CTF_VERSION_2`	3	Second version, with many range limits lifted.
`CTF_VERSION_3`	4	Third and current version, documented here.

This section documents `CTF_VERSION_3`.

2.1.1 CTF file-wide flags

The preamble contains bitflags in its `ctp_flags` field that describe various file-wide properties. Some of the flags are valid only for particular file-format versions, which means the flags can be used to fix file-format bugs. Consumers that see unknown flags should accordingly assume that the dictionary is not comprehensible, and refuse to open them.

The following flags are currently defined. Many are bug workarounds, valid only in CTFv3, and will not be valid in any future versions: the same values may be reused for other flags in v4+.

Flag	Versions	Value	Meaning
`CTF_F_COMPRESS`	All	0x1	Compressed with zlib
`CTF_F_NEWFUNCINFO`	3 only	0x2	"New-format" func info section.
`CTF_F_IDXSORTED`	3+	0x4	The index section is in sorted order
`CTF_F_DYNSTR`	3 only	0x8	The external strtab is in `.dynstr` and the symtab used is `.dynsym`. See Section 2.7 [The string section], page 20

`CTF_F_NEWFUNCINFO` and `CTF_F_IDXSORTED` relate to the function info and data object sections. See Section 2.4 [The symtypetab sections], page 19.

Further flags (and further compression methods) wil be added in future.

2.2 CTF header

The CTF header is the first part of a CTF dictionary, including the preamble. All parts of it other than the preamble (see Section 2.1 [CTF Preamble], page 4) can vary between CTF file versions and are never compressed. It contains things that apply to the dictionary as a whole, and a table of the sections into which the rest of the dictionary is divided. The sections tile the file: each section runs from the offset given until the start of the next section. Only the last section cannot follow this rule, so the header has a length for it instead.

All section offsets, here and in the rest of the CTF file, are relative to the *end* of the header. (This is annoyingly different to how offsets in CTF archives are handled.)

This is the first structure to include offsets into the string table, which are not straight references because CTF dictionaries can include references into the ELF string table to save space, as well as into the string table internal to the CTF dictionary. See Section 2.7 [The string section], page 20 for more on these. Offset 0 is always the null string.

```
typedef struct ctf_header
{
  ctf_preamble_t cth_preamble;
  uint32_t cth_parlabel;
  uint32_t cth_parname;
  uint32_t cth_cuname;
  uint32_t cth_lbloff;
  uint32_t cth_objtoff;
  uint32_t cth_funcoff;
  uint32_t cth_objtidxoff;
```

```
  uint32_t cth_funcidxoff;
  uint32_t cth_varoff;
  uint32_t cth_typeoff;
  uint32_t cth_stroff;
  uint32_t cth_strlen;
} ctf_header_t;
```

In detail:

Offset	Name	Description
0x00	`ctf_preamble_t cth_preamble`	The preamble (conceptually embedded in the header). See Section 2.1 [CTF Preamble], page 4
0x04	`uint32_t cth_parlabel`	The parent label, if deduplication happened against a specific label: a strtab offset. See Section 2.6 [The label section], page 20. Currently unused and always 0, but may be used in future when semantics are attached to the label section.
0x08	`uint32_t cth_parname`	The name of the parent dictionary deduplicated against: a strtab offset. Interpretation is up to the consumer (usually a CTF archive member name). 0 (the null string) if this is not a child dictionary.
0x1c	`uint32_t cth_cuname`	The name of the compilation unit, for consumers like GDB that want to know the name of CUs associated with single CUs: a strtab offset. 0 if this dictionary describes types from many CUs.
0x10	`uint32_t cth_lbloff`	The offset of the label section, which tiles the type space into named regions. See Section 2.6 [The label section], page 20.
0x14	`uint32_t cth_objtoff`	The offset of the data object symtypetab section, which maps ELF data symbols to types. See Section 2.4 [The symtypetab sections], page 19.
0x18	`uint32_t cth_funcoff`	The offset of the function info symtypetab section, which maps ELF function symbols to a return type and arg types. See Section 2.4 [The symtypetab sections], page 19.
0x1c	`uint32_t cth_objtidxoff`	The offset of the object index section, which maps ELF object symbols to entries in the data object section. See Section 2.4 [The symtypetab sections], page 19.

0x20	`uint32_t cth_funcidxoff`	The offset of the function info index section, which maps ELF function symbols to entries in the function info section. See Section 2.4 [The symtypetab sections], page 19.
0x24	`uint32_t cth_varoff`	The offset of the variable section, which maps string names to types. See Section 2.5 [The variable section], page 19.
0x28	`uint32_t cth_typeoff`	The offset of the type section, the core of CTF, which describes types using variable-length array elements. See Section 2.3 [The type section], page 7.
0x2c	`uint32_t cth_stroff`	The offset of the string section. See Section 2.7 [The string section], page 20.
0x30	`uint32_t cth_strlen`	The length of the string section (not an offset!). The CTF file ends at this point.

Everything from this point on (until the end of the file at `cth_stroff + cth_strlen`) is compressed with zlib if `CTF_F_COMPRESS` is set in the preamble's `ctp_flags`.

2.3 The type section

This section is the most important section in CTF, describing all the top-level types in the program. It consists of an array of type structures, each of which describes a type of some *kind*: each kind of type has some amount of variable-length data associated with it (some kinds have none). The amount of variable-length data associated with a given type can be determined by inspecting the type, so the reading code can walk through the types in sequence at opening time.

Each type structure is one of a set of overlapping structures in a discriminated union of sorts: the variable-length data for each type immediately follows the type's type structure. Here's the largest of the overlapping structures, which is only needed for huge types and so is very rarely seen:

```
typedef struct ctf_type
{
  uint32_t ctt_name;
  uint32_t ctt_info;
  __extension__
  union
  {
    uint32_t ctt_size;
    uint32_t ctt_type;
  };
  uint32_t ctt_lsizehi;
  uint32_t ctt_lsizelo;
} ctf_type_t;
```

Here's the much more common smaller form:

```
typedef struct ctf_stype
{
  uint32_t ctt_name;
  uint32_t ctt_info;
  __extension__
  union
  {
    uint32_t ctt_size;
    uint32_t ctt_type;
  };
} ctf_type_t;
```

If `ctt_size` is the #define `CTF_LSIZE_SENT`, 0xffffffff, this type is described by a `ctf_type_t`: otherwise, a `ctf_stype_t`.

Here's what the fields mean:

Offset	Name	Description
0x00	`uint32_t ctt_name`	Strtab offset of the type name, if any (0 if none).
0x04	`uint32_t ctt_info`	The *info word*, containing information on the kind of this type, its variable-length data and whether it is visible to name lookup. See See Section 2.3.1 [The info word], page 9.
0x08	`uint32_t ctt_size`	The size of this type, if this type is of a kind for which a size needs to be recorded (constant-size types don't need one). If this is `CTF_LSIZE_SENT`, this type is a huge type described by `ctf_type_t`.
0x08	`uint32_t ctt_type`	The type this type refers to, if this type is of a kind which refers to other types (like a pointer). All such types are fixed-size, and no types that are variable-size refer to other types, so `ctt_size` and `ctt_type` overlap. All type kinds that use `ctt_type` are described by `ctf_stype_t`, not `ctf_type_t`. See Section 2.3.2 [Type indexes and type IDs], page 9.
0x0c (`ctf_type_t` only)	`uint32_t ctt_lsizehi`	The high 32 bits of the size of a very large type. The `CTF_TYPE_LSIZE` macro can be used to get a 64-bit size out of this field and the next one. `CTF_SIZE_TO_LSIZE_HI` splits the `ctt_lsizehi` out of it again.

0x10 (`ctf_type_t` only)	`uint32_t ctt_lsizelo`	The low 32 bits of the size of a very large type. `CTF_SIZE_TO_LSIZE_LO` splits the `ctt_lsizelo` out of a 64-bit size.

Two aspects of this need further explanation: the info word, and what exactly a type ID is and how you determine it. (Information on the various type-kind- dependent things, like whether `ctt_size` or `ctt_type` is used, is described in the section devoted to each kind.)

2.3.1 The info word, ctt_info

The info word is a bitfield split into three parts. From MSB to LSB:

Bit offset	Name	Description
26–31	`kind`	Type kind: see Section 2.3.3 [Type kinds], page 10.
25	`isroot`	1 if this type is visible to name lookup
0–24	`vlen`	Length of variable-length data for this type (some kinds only). The variable-length data directly follows the `ctf_type_t` or `ctf_stype_t`. This is a kind-dependent array length value, not a length in bytes. Some kinds have no variable-length data, or fixed-size variable-length data, and do not use this value.

The most mysterious of these is undoubtedly `isroot`. This indicates whether types with names (nonzero `ctt_name`) are visible to name lookup: if zero, this type is considered a *non-root type* and you can't look it up by name at all. Multiple types with the same name in the same C namespace (struct, union, enum, other) can exist in a single dictionary, but only one of them may have a nonzero value for `isroot`. `libctf` validates this at open time and refuses to open dictionaries that violate this constraint.

Historically, this feature was introduced for the encoding of bitfields (see Section 2.3.4 [Integer types], page 12): for instance, int bitfields will all be named `int` with different widths or offsets, but only the full-width one at offset zero is wanted when you look up the type named `int`. With the introduction of slices (see Section 2.3.6 [Slices], page 14) as a more general bitfield encoding mechanism, this is less important, but we still use non-root types to handle conflicts if the linker API is used to fuse multiple translation units into one dictionary and those translation units contain types with the same name and conflicting definitions. (We do not discuss this further here, because the linker never does this: only specialized type mergers do, like that used for the Linux kernel. The libctf documentation will describe this in more detail.)

The `CTF_TYPE_INFO` macro can be used to compose an info word from a `kind`, `isroot`, and `vlen`; `CTF_V2_INFO_KIND`, `CTF_V2_INFO_ISROOT` and `CTF_V2_INFO_VLEN` pick it apart again.

2.3.2 Type indexes and type IDs

Types are referred to within the CTF file via *type IDs*. A type ID is a number from 0 to 2^32, from a space divided in half. Types $2^31 - 1$ and below are in the *parent range*: these IDs are used for dictionaries that have not had any other dictionary `ctf_import`ed into it as a parent. Both completely standalone dictionaries and parent dictionaries with children

hanging off them have types in this range. Types 2^31 and above are in the *child range*: only types in child dictionaries are in this range.

These IDs appear in `ctf_type_t.ctt_type` (see Section 2.3 [The type section], page 7), but the types themselves have no visible ID: quite intentionally, because adding an ID uses space, and every ID is different so they don't compress well. The IDs are implicit: at open time, the consumer walks through the entire type section and counts the types in the type section. The type section is an array of variable-length elements, so each entry could be considered as having an index, starting from 1. We count these indexes and associate each with its corresponding `ctf_type_t` or `ctf_stype_t`.

Lookups of types with IDs in the parent space look in the parent dictionary if this dictionary has one associated with it; lookups of types with IDs in the child space error out if the dictionary does not have a parent, and otherwise convert the ID into an index by shaving off the top bit and look up the index in the child.

These properties mean that the same dictionary can be used as a parent of child dictionaries and can also be used directly with no children at all, but a dictionary created as a child dictionary must always be associated with a parent — usually, the same parent — because its references to its own types have the high bit turned on and this is only flipped off again if this is a child dictionary. (This is not a problem, because if you *don't* associate the child with a parent, any references within it to its parent types will fail, and there are almost certain to be many such references, or why is it a child at all?)

This does mean that consumers should keep a close eye on the distinction between type IDs and type indexes: if you mix them up, everything will appear to work as long as you're only using parent dictionaries or standalone dictionaries, but as soon as you start using children, everything will fail horribly.

Type index zero, and type ID zero, are used to indicate that this type cannot be represented in CTF as currently constituted: they are emitted by the compiler, but all type chains that terminate in the unknown type are erased at link time (structure fields that use them just vanish, etc). So you will probably never see a use of type zero outside the symtypetab sections, where they serve as sentinels of sorts, to indicate symbols with no associated type.

The macros `CTF_V2_TYPE_TO_INDEX` and `CTF_V2_INDEX_TO_TYPE` may help in translation between types and indexes: `CTF_V2_TYPE_ISPARENT` and `CTF_V2_TYPE_ISCHILD` can be used to tell whether a given ID is in the parent or child range.

It is quite possible and indeed common for type IDs to point forward in the dictionary, as well as backward.

2.3.3 Type kinds

Every type in CTF is of some *kind*. Each kind is some variety of C type: all structures are a single kind, as are all unions, all pointers, all arrays, all integers regardless of their bitfield width, etc. The kind of a type is given in the `kind` field of the `ctt_info` word (see Section 2.3.1 [The info word], page 9).

The space of type kinds is only a quarter full so far, so there is plenty of room for expansion. It is likely that in future versions of the file format, types with smaller kinds will be more efficiently encoded than types with larger kinds, so their numerical value will actually start to matter in future. (So these IDs will probably change their numerical values

in a later release of this format, to move more frequently-used kinds like structures and cv-quals towards the top of the space, and move rarely-used kinds like integers downwards. Yes, integers are rare: how many kinds of `int` are there in a program? They're just very frequently *referenced*.)

Here's the set of kinds so far. Each kind has a `#define` associated with it, also given here.

Kind	Macro	Purpose
0	`CTF_K_UNKNOWN`	Indicates a type that cannot be represented in CTF, or that is being skipped. It is very similar to type ID 0, except that you can have *multiple*, distinct types of kind `CTF_K_UNKNOWN`.
1	`CTF_K_INTEGER`	An integer type. See Section 2.3.4 [Integer types], page 12.
2	`CTF_K_FLOAT`	A floating-point type. See Section 2.3.5 [Floating-point types], page 13.
3	`CTF_K_POINTER`	A pointer. See Section 2.3.7 [Pointers typedefs and cvr-quals], page 15.
4	`CTF_K_ARRAY`	An array. See Section 2.3.8 [Arrays], page 15.
5	`CTF_K_FUNCTION`	A function pointer. See Section 2.3.9 [Function pointers], page 16.
6	`CTF_K_STRUCT`	A structure. See Section 2.3.11 [Structs and unions], page 17.
7	`CTF_K_UNION`	A union. See Section 2.3.11 [Structs and unions], page 17.
8	`CTF_K_ENUM`	An enumerated type. See Section 2.3.10 [Enums], page 16.
9	`CTF_K_FORWARD`	A forward. See Section 2.3.12 [Forward declarations], page 18.
10	`CTF_K_TYPEDEF`	A typedef. See Section 2.3.7 [Pointers typedefs and cvr-quals], page 15.
11	`CTF_K_VOLATILE`	A volatile-qualified type. See Section 2.3.7 [Pointers typedefs and cvr-quals], page 15.
12	`CTF_K_CONST`	A const-qualified type. See Section 2.3.7 [Pointers typedefs and cvr-quals], page 15.
13	`CTF_K_RESTRICT`	A restrict-qualified type. See Section 2.3.7 [Pointers typedefs and cvr-quals], page 15.

14	`CTF_K_SLICE`	A slice, a change of the bit-width or offset of some other type. See Section 2.3.6 [Slices], page 14.

Now we cover all type kinds in turn. Some are more complicated than others.

2.3.4 Integer types

Integral types are all represented as types of kind `CTF_K_INTEGER`. These types fill out `ctt_size` in the `ctf_stype_t` with the size in bytes of the integral type in question. They are always represented by `ctf_stype_t`, never `ctf_type_t`. Their variable-length data is one `uint32_t` in length: `vlen` in the info word should be disregarded and is always zero.

The variable-length data for integers has multiple items packed into it much like the info word does.

Bit offset	Name	Description
24–31	Encoding	The desired display representation of this integer. You can extract this field with the `CTF_INT_ENCODING` macro. See below.
16–23	Offset	The offset of this integral type in bits from the start of its enclosing structure field, adjusted for endianness: see Section 2.3.11 [Structs and unions], page 17. You can extract this field with the `CTF_INT_OFFSET` macro.
0–15	Bit-width	The width of this integral type in bits. You can extract this field with the `CTF_INT_BITS` macro.

If you choose, bitfields can be represented using the things above as a sort of integral type with the `isroot` bit flipped off and the offset and bits values set in the vlen word: you can populate it with the `CTF_INT_DATA` macro. (But it may be more convenient to represent them using slices of a full-width integer: see Section 2.3.6 [Slices], page 14.)

Integers that are bitfields usually have a `ctt_size` rounded up to the nearest power of two in bytes, for natural alignment (e.g. a 17-bit integer would have a `ctt_size` of 4). However, not all types are naturally aligned on all architectures: packed structures may in theory use integral bitfields with different `ctt_size`, though this is rarely observed.

The *encoding* for integers is a bit-field comprised of the values below, which consumers can use to decide how to display values of this type:

Offset	Name	Description
0x01	`CTF_INT_SIGNED`	If set, this is a signed int: if false, unsigned.
0x02	`CTF_INT_CHAR`	If set, this is a char type. It is platform-dependent whether unadorned `char` is signed or not: the `CTF_CHAR` macro produces an integral type suitable for the definition of `char` on this platform.
0x04	`CTF_INT_BOOL`	If set, this is a boolean type. (It is theoretically possible to turn this and `CTF_INT_CHAR` on at the same time, but it is not clear what this would mean.)

0x08	`CTF_INT_VARARGS`	If set, this is a varargs-promoted value in a K&R function definition. This is not currently produced or consumed by anything that we know of: it is set aside for future use.

The GCC "`Complex int`" and fixed-point extensions are not yet supported: references to such types will be emitted as type 0.

2.3.5 Floating-point types

Floating-point types are all represented as types of kind `CTF_K_FLOAT`. Like integers, These types fill out `ctt_size` in the `ctf_stype_t` with the size in bytes of the floating-point type in question. They are always represented by `ctf_stype_t`, never `ctf_type_t`.

This part of CTF shows many rough edges in the more obscure corners of floating-point handling, and is likely to change in format v4.

The variable-length data for floats has multiple items packed into it just like integers do:

Bit offset	Name	Description
24–31	Encoding	The desired display representation of this float. You can extract this field with the `CTF_FP_ENCODING` macro. See below.
16–23	Offset	The offset of this floating-point type in bits from the start of its enclosing structure field, adjusted for endianness: see Section 2.3.11 [Structs and unions], page 17. You can extract this field with the `CTF_FP_OFFSET` macro.
0–15	Bit-width	The width of this floating-point type in bits. You can extract this field with the `CTF_FP_BITS` macro.

The purpose of the floating-point offset and bit-width is somewhat opaque, since there are no such things as floating-point bitfields in C: the bit-width should be filled out with the full width of the type in bits, and the offset should always be zero. It is likely that these fields will go away in the future. As with integers, you can use `CTF_FP_DATA` to assemble one of these vlen items from its component parts.

The *encoding* for floats is not a bitfield but a simple value indicating the display representation. Many of these are unused, relate to Solaris-specific compiler extensions, and will be recycled in future: some are unused and will become used in future.

Offset	Name	Description
1	`CTF_FP_SINGLE`	This is a single-precision IEEE 754 `float`.
2	`CTF_FP_DOUBLE`	This is a double-precision IEEE 754 `double`.
3	`CTF_FP_CPLX`	This is a `Complex float`.
4	`CTF_FP_DCPLX`	This is a `Complex double`.
5	`CTF_FP_LDCPLX`	This is a `Complex long double`.
6	`CTF_FP_LDOUBLE`	This is a `long double`.
7	`CTF_FP_INTRVL`	This is a `float` interval type, a Solaris-specific extension. Unused: will be recycled.
8	`CTF_FP_DINTRVL`	This is a `double` interval type, a Solaris-specific extension. Unused: will be recycled.

9	`CTF_FP_LDINTRVL`	This is a `long double` interval type, a Solaris-specific extension. Unused: will be recycled.
10	`CTF_FP_IMAGRY`	This is a the imaginary part of a `Complex float`. Not currently generated. May change.
11	`CTF_FP_DIMAGRY`	This is a the imaginary part of a `Complex double`. Not currently generated. May change.
12	`CTF_FP_LDIMAGRY`	This is a the imaginary part of a `Complex long double`. Not currently generated. May change.

The use of the complex floating-point encodings is obscure: it is possible that `CTF_FP_CPLX` is meant to be used for only the real part of complex types, and `CTF_FP_IMAGRY` et al for the imaginary part – but for now, we are emitting `CTF_FP_CPLX` to cover the entire type, with no way to get at its constituent parts. There appear to be no uses of these encodings anywhere, so they are quite likely to change incompatibly in future.

2.3.6 Slices

Slices, with kind `CTF_K_SLICE`, are an unusual CTF construct: they do not directly correspond to any C type, but are a way to model other types in a more convenient fashion for CTF generators.

A slice is like a pointer or other reference type in that they are always represented by `ctf_stype_t`: but unlike pointers and other reference types, they populate the `ctt_size` field just like integral types do, and come with an attached encoding and transform the encoding of the underlying type. The underlying type is described in the variable-length data, similarly to structure and union fields: see below. Requests for the type size should also chase down to the referenced type.

Slices are always nameless: `ctt_name` is always zero for them.

(The `libctf` API behaviour is unusual as well, and justifies the existence of slices: `ctf_type_kind` never returns `CTF_K_SLICE` but always the underlying type kind, so that consumers never need to know about slices: they can tell if an apparent integer is actually a slice if they need to by calling `ctf_type_reference`, which will uniquely return the underlying integral type rather than erroring out with `ECTF_NOTREF` if this is actually a slice. So slices act just like an integer with an encoding, but more closely mirror DWARF and other debugging information formats by allowing CTF file creators to represent a bitfield as a slice of an underlying integral type.)

The vlen in the info word for a slice should be ignored and is always zero. The variable-length data for a slice is a single `ctf_slice_t`:

```
typedef struct ctf_slice
{
  uint32_t cts_type;
  unsigned short cts_offset;
  unsigned short cts_bits;
} ctf_slice_t;
```

Offset	Name	Description

0x0	`uint32_t cts_type`	The type this slice is a slice of. Must be an integral type (or a floating-point type, but this nonsensical option will go away in v4.)
0x4	`unsigned short cts_offset`	The offset of this integral type in bits from the start of its enclosing structure field, adjusted for endianness: see Section 2.3.11 [Structs and unions], page 17. Identical semantics to the `CTF_INT_OFFSET` field: see Section 2.3.4 [Integer types], page 12. This field is much too long, because the maximum possible offset of an integral type would easily fit in a char: this field is bigger just for the sake of alignment. This will change in v4.
0x6	`unsigned short cts_bits`	The bit-width of this integral type. Identical semantics to the `CTF_INT_BITS` field: see Section 2.3.4 [Integer types], page 12. As above, this field is really too large and will shrink in v4.

2.3.7 Pointers, typedefs, and cvr-quals

Pointers, `typedef`s, and `const`, `volatile` and `restrict` qualifiers are represented identically except for their type kind (though they may be treated differently by consuming libraries like `libctf`, since pointers affect assignment-compatibility in ways cvr-quals do not, and they may have different alignment requirements, etc).

All of these are represented by `ctf_stype_t`, have no variable data at all, and populate `ctt_type` with the type ID of the type they point to. These types can stack: a `CTF_K_RESTRICT` can point to a `CTF_K_CONST` which can point to a `CTF_K_POINTER` etc.

They are all unnamed: `ctt_name` is 0.

The size of `CTF_K_POINTER` is derived from the data model (see Section 2.8 [Data models], page 21), i.e. in practice, from the target machine ABI, and is not explicitly represented. The size of other kinds in this set should be determined by chasing ctf_types as necessary until a non-typedef/const/volatile/restrict is found, and using that.

2.3.8 Arrays

Arrays are encoded as types of kind `CTF_K_ARRAY` in a `ctf_stype_t`. Both size and kind for arrays are zero. The variable-length data is a `ctf_array_t`: `vlen` in the info word should be disregarded and is always zero.

```
typedef struct ctf_array
{
  uint32_t cta_contents;
  uint32_t cta_index;
  uint32_t cta_nelems;
} ctf_array_t;
```

Offset	Name	Description
0x0	`uint32_t cta_contents`	The type of the array elements: a type ID.
0x4	`uint32_t cta_index`	The type of the array index: a type ID of an integral type. If this is a variable-length array, the index type ID will be 0 (but the actual index type of this array is probably `int`). Probably redundant and may be dropped in v4.
0x8	`uint32_t cta_nelems`	The number of array elements. 0 for VLAs, and also for the historical variety of VLA which has explicit zero dimensions (which will have a nonzero `cta_index`.)

The size of an array can be computed by simple multiplication of the size of the `cta_contents` type by the `cta_nelems`.

2.3.9 Function pointers

Function pointers are explicitly represented in the CTF type section by a type of kind `CTF_K_FUNCTION`, always encoded with a `ctf_stype_t`. The `ctt_type` is the function return type ID. The `vlen` in the info word is the number of arguments, each of which is a type ID, a `uint32_t`: if the last argument is 0, this is a varargs function and the number of arguments is one less than indicated by the vlen.

If the number of arguments is odd, a single `uint32_t` of padding is inserted to maintain alignment.

2.3.10 Enums

Enumerated types are represented as types of kind `CTF_K_ENUM` in a `ctf_stype_t`. The `ctt_size` is always the size of an int from the data model (enum bitfields are implemented via slices). The `vlen` is a count of enumerations, each of which is represented by a `ctf_enum_t` in the vlen:

```
typedef struct ctf_enum
{
  uint32_t cte_name;
  int32_t cte_value;
} ctf_enum_t;
```

Offset	Name	Description
0x0	`uint32_t cte_name`	Strtab offset of the enumeration name. Must not be 0.
0x4	`int32_t cte_value`	The enumeration value.

Enumeration values larger than 2^32 are not yet supported and are omitted from the enumeration. (v4 will lift this restriction by encoding the value differently.)

Forward declarations of enums are not implemented with this kind: see Section 2.3.12 [Forward declarations], page 18.

Enumerated type names, as usual in C, go into their own namespace, and do not conflict with non-enums, structs, or unions with the same name.

2.3.11 Structs and unions

Structures and unions are represnted as types of kind `CTF_K_STRUCT` and `CTF_K_UNION`: their representation is otherwise identical, and it is perfectly allowed for "structs" to contain overlapping fields etc, so we will treat them together for the rest of this section.

They fill out `ctt_size`, and use `ctf_type_t` in preference to `ctf_stype_t` if the structure size is greater than `CTF_MAX_SIZE` (0xfffffffe).

The vlen for structures and unions is a count of structure fields, but the type used to represent a structure field (and thus the size of the variable-length array element representing the type) depends on the size of the structure: truly huge structures, greater than `CTF_LSTRUCT_THRESH` bytes in length, use a different type. (`CTF_LSTRUCT_THRESH` is 536870912, so such structures are vanishingly rare: in v4, this representation will change somewhat for greater compactness. It's inherited from v1, where the limits were much lower.)

Most structures can get away with using `ctf_member_t`:

```
typedef struct ctf_member_v2
{
  uint32_t ctm_name;
  uint32_t ctm_offset;
  uint32_t ctm_type;
} ctf_member_t;
```

Huge structures that are represented by `ctf_type_t` rather than `ctf_stype_t` have to use `ctf_lmember_t`, which splits the offset as `ctf_type_t` splits the size:

```
typedef struct ctf_lmember_v2
{
  uint32_t ctlm_name;
  uint32_t ctlm_offsethi;
  uint32_t ctlm_type;
  uint32_t ctlm_offsetlo;
} ctf_lmember_t;
```

Here's what the fields of `ctf_member` mean:

Offset	Name	Description
0x00	`uint32_t ctm_name`	Strtab offset of the field name.
0x04	`uint32_t ctm_offset`	The offset of this field *in bits*. (Usually, for bitfields, this is machine-word-aligned and the individual field has an offset in bits, but the format allows for the offset to be encoded in bits here.)
0x08	`uint32_t ctm_type`	The type ID of the type of the field.

Here's what the fields of the very similar `ctf_lmember` mean:

Offset	Name	Description
0x00	`uint32_t ctlm_name`	Strtab offset of the field name.
0x04	`uint32_t ctlm_offsethi`	The high 32 bits of the offset of this field in bits.
0x08	`uint32_t ctlm_type`	The type ID of the type of the field.
0x0c	`uint32_t ctlm_offsetlo`	The low 32 bits of the offset of this field in bits.

Macros `CTF_LMEM_OFFSET`, `CTF_OFFSET_TO_LMEMHI` and `CTF_OFFSET_TO_LMEMLO` serve to extract and install the values of the `ctlm_offset` fields, much as with the split size fields in `ctf_type_t`.

Unnamed structure and union fields are simply implemented by collapsing the unnamed field's members into the containing structure or union: this does mean that a structure containing an unnamed union can end up being a "structure" with multiple members at the same offset. (A future format revision may collapse `CTF_K_STRUCT` and `CTF_K_UNION` into the same kind and decide among them based on whether their members do in fact overlap.)

Structure and union type names, as usual in C, go into their own namespace, just as enum type names do.

Forward declarations of structures and unions are not implemented with this kind: see Section 2.3.12 [Forward declarations], page 18.

2.3.12 Forward declarations

When the compiler encounters a forward declaration of a struct, union, or enum, it emits a type of kind `CTF_K_FORWARD`. If it later encounters a non- forward declaration of the same thing, it marks the forward as non-root-visible: before link time, therefore, non-root-visible forwards indicate that a non-forward is coming.

After link time, forwards are fused with their corresponding non-forwards by the deduplicator where possible. They are kept if there is no non-forward definition (maybe it's not visible from any TU at all) or if `multiple` conflicting structures with the same name might match it. Otherwise, all other forwards are converted to structures, unions, or enums as appropriate, even across TUs if only one structure could correspond to the forward (after all, all types across all TUs land in the same dictionary unless they conflict, so promoting forwards to their concrete type seems most helpful).

A forward has a rather strange representation: it is encoded with a `ctf_stype_t` but the `ctt_type` is populated not with a type (if it's a forward, we don't have an underlying type yet: if we did, we'd have promoted it and this wouldn't be a forward any more) but with the `kind` of the forward. This means that we can distinguish forwards to structs, enums and unions reliably and ensure they land in the appropriate namespace even before the actual struct, union or enum is found.

2.4 The symtypetab sections

These are two very simple sections with identical formats, used by consumers to map from ELF function and data symbols directly to their types. So they are usually populated only in CTF sections that are embedded in ELF objects.

Their format is very simple: an array of type IDs. Which symbol each type ID corresponds to depends on whether the optional *index section* associated with this symtypetab section has any content.

If the index section is nonempty, it is an array of `uint32_t` string table offsets, each giving the name of the symbol whose type is at the same offset in the corresponding non-index section: users can look up symbols in such a table by name. The index section and corresponding symtypetab section is usually ASCIIbetically sorted (indicated by the `CTF_F_IDXSORTED` flag in the header): if it's sorted, it can be bsearched for a symbol name rather than having to use a slower linear search.

If the data object index section is empty, the entries in the data object and function info sections are associated 1:1 with ELF symbols of type `STT_OBJECT` (for data object) or `STT_FUNC` (for function info) with a nonzero value: the linker shuffles the symtypetab sections to correspond with the order of the symbols in the ELF file. Symbols with no name, undefined symbols and symbols named "`_START_`" and "`_END_`" are skipped and never appear in either section. Symbols that have no corresponding type are represented by type ID 0. The section may have fewer entries than the symbol table, in which case no later entries have associated types. This format is more compact than an indexed form if most entries have types (since there is no need to record any symbol names), but if the producer and consumer disagree even slightly about which symbols are omitted, the types of all further symbols will be wrong!

The compiler always emits indexed symtypetab tables, because there is no symbol table yet. The linker will always have to read them all in and always works through them from start to end, so there is no benefit having the compiler sort them either. The linker (actually, `libctf`'s linking machinery) will automatically sort unsorted indexed sections, and convert indexed sections that contain a lot of pads into the more compact, unindexed form.

If child dicts are in use, only symbols that use types actually mentioned in the child appear in the child's symtypetab: symbols that use only types in the parent appear in the parent's symtypetab instead. So the child's symtypetab will almost always be very sparse, and thus will usually use the indexed form even in fully linked objects. (It is, of course, impossible for symbols to exist that use types from multiple child dicts at once, since it's impossible to declare a function in C that uses types that are only visible in two different, disjoint translation units.)

2.5 The variable section

The variable section is a simple array mapping names (strtab entries) to type IDs, intended to provide a replacement for the data object section in dynamic situations in which there is no static ELF strtab but the consumer instead hands back names. The section is sorted into ASCIIbetical order by name for rapid lookup, like the CTF archive name table.

The section is an array of these structures:

```
typedef struct ctf_varent
```

```
{
  uint32_t ctv_name;
  uint32_t ctv_type;
} ctf_varent_t;
```

Offset	Name	Description
0x00	`uint32_t ctv_name`	Strtab offset of the name
0x04	`uint32_t ctv_type`	Type ID of this type

There is no analogue of the function info section yet: v4 will probably drop this section in favour of a way to put both indexed (thus, named) and nonindexed symbols into the symtypetab sections at the same time.

2.6 The label section

The label section is a currently-unused facility allowing the tiling of the type space with names taken from the strtab. The section is an array of these structures:

```
typedef struct ctf_lblent
{
  uint32_t ctl_label;
  uint32_t ctl_type;
} ctf_lblent_t;
```

Offset	Name	Description
0x00	`uint32_t ctl_label`	Strtab offset of the label
0x04	`uint32_t ctl_type`	Type ID of the last type covered by this label

Semantics will be attached to labels soon, probably in v4 (the plan is to use them to allow multiple disjoint namespaces in a single CTF file, removing many uses of CTF archives, in particular in the `.ctf` section in ELF objects).

2.7 The string section

This section is a simple ELF-format strtab, starting with a zero byte (thus ensuring that the string with offset 0 is the null string, as assumed elsewhere in this spec). The strtab is usually ASCIIbetically sorted to somewhat improve compression efficiency.

Where the strtab is unusual is the *references* to it. CTF has two string tables, the internal strtab and an external strtab associated with the CTF dictionary at open time: usually, this is the ELF dynamic strtab (`.dynstr`) of a CTF dictionary embedded in an ELF file. We distinguish between these strtabs by the most significant bit, bit 31, of the 32-bit strtab references: if it is 0, the offset is in the internal strtab: if 1, the offset is in the external strtab.

There is a bug workaround in this area: in format v3 (the first version to have working support for external strtabs), the external strtab is `.strtab` unless the `CTF_F_DYNSTR` flag is set on the dictionary (see Section 2.1.1 [CTF file-wide flags], page 5). Format v4

will introduce a header field that explicitly names the external strtab, making this flag unnecessary.

2.8 Data models

The data model is a simple integer which indicates the ABI in use on this platform. Right now, it is very simple, distinguishing only between 32- and 64-bit types: a model of 1 indicates ILP32, 2 indicats LP64. The mapping from ABI integer to type sizes is hardwired into `libctf`: currently, we use this to hardwire the size of pointers, function pointers, and enumerated types,

This is a very kludgy corner of CTF and will probably be replaced with explicit header fields to record this sort of thing in future.

2.9 Limits of CTF

The following limits are imposed by various aspects of CTF version 3:

`CTF_MAX_TYPE`
: Maximum type identifier (maximum number of types accessible with parent and child containers in use): 0xfffffffe

`CTF_MAX_PTYPE`
: Maximum type identifier in a parent dictioanry: maximum number of types in any one dictionary: 0x7fffffff

`CTF_MAX_NAME`
: Maximum offset into a string table: 0x7fffffff

`CTF_MAX_VLEN`
: Maximum number of members in a struct, union, or enum: maximum number of function args: 0xffffff

`CTF_MAX_SIZE`
: Maximum size of a `ctf_stype_t` in bytes before we fall back to `ctf_type_t`: 0xfffffffe bytes

Other maxima without associated macros:

- Maximum value of an enumerated type: 2^32
- Maximum size of an array element: 2^32

These maxima are generally considered to be too low, because C programs can and do exceed them: they will be lifted in format v4.

Index

A

B

C

D

E

F

I

L

U

V