NAME

ctf – Compact C Type Format

SYNOPSIS

#include <ctf.h>

DESCRIPTION

CTF (Compact C Type Format) is designed to be a compact representation of the C programming language's type information, focused on serving the needs of dynamic tracing, debuggers, and other in-situ and post-mortem introspection tools. **CTF** data is generally included in **ELF** objects, in a section named **.SUNW_ctf**, of type **SHT_SUNW_CTF**, to ensure that the data is accessible in a running process and in subsequent core dumps, if generated.

The **CTF** data contained in each file has information about the layout and sizes of C types, including intrinsic types, enumerations, structures, typedefs, and unions, that are used by the corresponding **ELF** object. The **CTF** data may also include information about the types of global objects and the return type and arguments of functions in the symbol table.

Because **CTF** is often embedded inside object files, rather than in a standalone file, it can also be referred to as a **container**.

On Oracle Solaris systems, **CTF** data is consumed by DTrace and mdb. See **dtrace**(8), and **mdb**(1). Programmatic access to **CTF** data is obtained through the shared object **libctf**.

The **CTF** file format consists of a header, followed by sections providing information for labels, objects (data), functions, types, and strings. The header starts with a **preamble** field that describes the version, followed by links to other files, and the offsets within the file at which the following sections can be accessed. The first section that follows the header is the **label** section, which provides a way of identifying similar groups of **CTF** data across multiple files. This is followed by the **object** information section, which describes the types of global symbols. The subsequent section is the **function** information section, which describes the return types and arguments of functions. The next section is the **type** information section, which describes the format and layout of the C types used in the object, and finally the last section is the **string** section, which contains the names of types, enumerations, members, and labels.

To be well formed, a **CTF** file need only contain the header, but to be minimally useful, the type and string sections must also be present.

A **CTF** file may contain the full set of type information required by the associated object, or it may optionally provide a subset, and reference another **CTF** file which provides the remaining types. When a **CTF** file refers to another file, it is called the **child**, and the file it refers to is called the **parent**. A given file may only refer to a single parent. This process is called **uniquification** because it ensures each child only has type information that is unique to it. A common example of this is that most Solaris kernel modules are uniquified against the **genunix** kernel module. This provides organizational and space saving benefits, as the **CTF** for each such module only describes the types that are unique to it, and the common types shared from the kernel are not duplicated.

FILE FORMAT

This document describes versions 2 and 3 of the **CTF** file format. The versions are very similar, offering the same features, and differing primarily in that version 3 is able to support a greatly expanded number of identifiers, and to describe larger types. All cases in which the two versions differ are explicitly noted. When no **CTF** version is specified, the description applies equally to both versions. The **ctfconvert**(1), **ctfmerge**(1), and **ld**(1) utilities are capable of producing either version 2 or 3. Unless explicitly requested, **CTF** version 3 is produced by default. The **libctf** library provides access to **CTF** data in a version independent manner. Applications that obtain **CTF** through the use of **libctf** are therefore able to read all versions

of **CTF**.

Every **CTF** file begins with a **header** that describes the remainder of the file. The first field in the header is the **preamble**, which is defined as follows.

```
typedef struct ctf_preamble {
        uint16_t ctp_magic;     /* magic number (CTF_MAGIC) */
        uint8_t ctp_version;    /* format version (CTF_VERSION) */
        uint8_t ctp_flags;      /* flags (see below) */
} ctf_preamble_t;
```

This **preamble**, which is four bytes long and must be four byte aligned, identifies a **CTF** file, and provides the format version employed. All versions of **CTF** are guaranteed to retain this definition of the preamble, and to place it at the top of the file, ensuring that all CTF reading applications can always identify all versions of CTF. The **ctp_version** and **ctp_flags** fields are single byte values, and can be read directly by both big and little endian machines without concern for byte order. The fields in a preamble are as follows:

ctp_magic

A **magic number** that serves to identify the file as a **CTF** container. Valid magic numbers are:

```
#define CTF_MAGIC       0xcff1 /* identifying magic number */
#define CTF_MAGIC_XLATE 0xf1cf /* CTF_MAGIC (rev. byte order) */
```

A value of **CTF_MAGIC** indicates a **CTF** file, while **CTF_MAGIC_XLATE** indicates a **CTF** file that was produced on a machine that employs the opposite byteorder from that of the examining machine. If another value is encountered, then the file should not be treated as **CTF** file.

ctp_version

The **CTF** format version. The following versions are currently defined.

```
#define CTF_VERSION_1   1
#define CTF_VERSION_2   2
#define CTF_VERSION_3   3
#define CTF_VERSION     CTF_VERSION_3   /* current version */
```

The current version is 3. It is possible to encounter an unsupported version. In that case, software should not try to parse the format, as it may have changed.

ctp_flags

ctp_flags describes aspects of the file which modify its interpretation. Flags are defined independently for each version, so in principle, the value of **ctp_version** must be taken into account when evaluating **ctp_flags**. However, all versions of **CTF** to date define a single flag which is the same in all versions:

```
#define CTF_F_COMPRESS  0x1     /* data buffer is compressed */
```

The flag **CTF_F_COMPRESS** indicates that the body of the file, all the data following the **header**

has been compressed using the **libz** compression library **deflate** algorithm. See **libz**(3). If this flag is not present, then the body has not been compressed and no special action is needed to interpret it. All offsets into the data, as described by the **CTF** header, always refer to the uncompressed data.

The first field in the header, for all **CTF** versions, is always the **preamble**, described above. The remainder of the header can differ for each version. However, versions 2 and 3 both use the following common definition. In addition to the preamble, this header describes whether or not the **CTF** file is the child of another **CTF** file, and provides the position and size of the sections that follow the header. The structure for the header starts with the preamble, and has an overall size of 36 bytes.

```
typedef struct ctf_header {
        ctf_preamble_t cth_preamble;
        uint32_t cth_parlabel;  /* ref to parent lbl uniq'd against */
        uint32_t cth_parname;   /* ref to basename of parent */
        uint32_t cth_lbloff;    /* offset of label section */
        uint32_t cth_objtoff;   /* offset of object section */
        uint32_t cth_funcoff;   /* offset of function section */
        uint32_t cth_typeoff;   /* offset of type section */
        uint32_t cth_stroff;    /* offset of string section */
        uint32_t cth_strlen;    /* length of string section in bytes */
} ctf_header_t;
```

After the preamble, the next two fields, **cth_parlabel** and **cth_parname**, are used to identify the parent. The value of both fields are offsets into the **string** section which point to the start of a null-terminated string. For more information on the encoding of strings, see **String Identifiers**. If the value of either is 0, then there is no entry for that field. If the field **cth_parlabel** is set, then the **ctf_parname** field must be set, otherwise it will not be possible to find the parent. If **ctf_parname** is set, it is not necessary to define **cth_parlabel**, as the parent may not have a label. For more information on labels and their interpretation, see **The Label Section**.

The remaining header fields, excepting **cth_strlen**, provide the offset of each section, relative to the end of the header. To convert such an offset into an offset taken relative to the top of the **CTF** file, it suffices to add the size of **ctf_header_t** (36 bytes) to the offset. To calculate the size of a given section, excepting the string section, subtract the offset of the section from the that of the following one. For example, the size of the type section can be calculated by subtracting **cth_typeoff** from **cth_stroff**.

The offsets reflect the alignment requirements for each section. The **cth_objtoff** and **cth_funcoff** sections must be two-byte aligned for **CTF** version 2, and four-byte aligned for version 3. The sections **cth_lbloff** and **cth_typeoff** must be four-byte aligned for both **CTF** versions. The section **cth_stroff** has no alignment requirements.

Finally, **cth_strlen** provides the length of the string section. Since the string section is the last section in a **CTF** file, its offset and size can be used to determine the uncompressed size of the entire **CTF** file: **sizeof (ctf_header_t) + hdr->cth_stroff + hdr->cth_strlen**.

Type Identifiers

CTF data types are referred to using integer values known as **type identifiers**. In all versions of **CTF**, type identifier 0 is a sentinel value used to indicate that there is no type information available or it is an unknown type. The first valid type identifier is 1. When a given **CTF** file is a child, indicated by a non-zero entry for the **cth_parname** field of the header, then the most significant bit of the type ID is set. Macros are provided for detecting whether a given ID is for a parent or child, and for manipulating the **child bit**.

Type identifiers are unsigned 16-bit values in **CTF** version 2, and 32-bit values in version 3. As such, their range, and the bit used to identify child types differ. However, the same general approach is used for both.

CTF Version 2

```
#define CTF_V2_TYPE_ISPARENT(_id)  ((_id) < 0x8000)
#define CTF_V2_TYPE_ISCHILD(_id)  ((_id) > 0x7fff)
#define CTF_V2_TYPE_TO_INDEX(_id) ((_id) & 0x7fff)
#define CTF_V2_INDEX_TO_TYPE(_id, _ischild) \
  ((_ischild) ? ((_id) | 0x8000) : (_id))
```

CTF version 2 uses 16-bit unsigned integers to represent type IDs, and supports up to 32767 (0x7fff) types. In child **CTF** files, the first valid type identifier is 0x8000 and the last is 0xffff. In this case, type identifiers 1 through 0x7fff are references to the parent.

CTF Version 3

```
#define CTF_V3_TYPE_ISPARENT(_id) ((uint32_t)(_id) < 0x80000000u)
#define CTF_V3_TYPE_ISCHILD(_id)  ((uint32_t)(_id) > 0x7fffffffu)
#define CTF_V3_TYPE_TO_INDEX(_id) ((_id) & 0x7fffffffu)
#define CTF_V3_INDEX_TO_TYPE(_id, _ischild) \
  (((_id) & 0x7fffffffu) | ((_ischild) != 0 ? 0x80000000u : 0))
```

CTF version 3 uses 32-bit unsigned integers to represent type IDs, and supports up to 2147483646 (0x7ffffffe) types. In child **CTF** files, the first valid type identifier is 0x80000000 and the last is 0xfffffffe. In this case, type identifiers 1 through 0x7ffffffe are references to the parent. 0x7fffffff and 0xffffffff are not treated as valid type identifiers so as to enable the use of -1 as an error value.

String Identifiers

String identifiers are encoded as 32-bit unsigned integers which specify an offset into a string table. The **CTF** format supports two different string tables which have an identifier of 0 or 1. This identifier is stored in the high-order bit of the offset. Therefore, the maximum supported offset into one of these tables is 0x7fffffff.

Table identifier 0 refers to the **string** section that is included with the **CTF** file, which is located using the **cth_stroff** field of the **CTF** header. String table identifier 1 refers to the **ELF** string table associated with the symbol table from the associated **ELF** object.

Type Encoding

Every **CTF** type begins with metadata encoded into an integer value. This value is a 16-bit unsigned integer in **CTF** version 2, and a 32-bit unsigned integer in **CTF** version 3. This encoded information provides the following information:

- The kind of the type.
- Whether this type is a root type or not.
- The length of the variable data.

CTF Version 2

The 16 bits that make up the encoding are broken down into five bits for the kind (bits 11 to 15), one bit for the root type flag (bit 10), and 10 bits for the length of the variable data.

```
+------+--------+------+
| kind | isroot | vlen |
+------+--------+------+
15   11    10    9     0
```

CTF Version 3

The 32 bits that make up the encoding are broken down into six bits for the kind (bits 26 to 31), one bit for the root type flag (bit 25), and 25 bits for the length of the variable data.

```
+--------+--------+----------------------------+
| kind   | isroot | vlen                       |
+--------+--------+----------------------------+
31     26    25    24                         0
```

CTF currently defines the following 15 different kinds. The interpretation of these different kinds is discussed in **The Type Section**. If a kind is encountered that is not in this list, the **CTF** file should be considered to be invalid.

```
#define CTF_K_UNKNOWN   0
#define CTF_K_INTEGER   1
#define CTF_K_FLOAT     2
#define CTF_K_POINTER   3
#define CTF_K_ARRAY     4
#define CTF_K_FUNCTION  5
#define CTF_K_STRUCT    6
#define CTF_K_UNION     7
#define CTF_K_ENUM      8
#define CTF_K_FORWARD   9
#define CTF_K_TYPEDEF   10
#define CTF_K_VOLATILE  11
#define CTF_K_CONST     12
#define CTF_K_RESTRICT  13
#define CTF_K_SLICE     14
```

Programs directly reference many types; however, other types are referenced indirectly because they are part of some other structure. Types that are referenced directly and used are called **root** types. Other types may be used indirectly, for example, a program may reference a structure directly, but not one of its members which has a type. That type is not considered a root type. If a type is a root type, then it will have the root flag set.

The meaning of the variable length (**vlen**) is specific to each kind and is discussed in **The Type Section**.

The following macros are useful for constructing and deconstructing the encoded **info** word. Note that there

are different macros for **CTF** version 2 and version 3, each used to encode the details of the info word for that version.

```
#define CTF_V2_MAX_VLEN         0x3ff /* max # variant data items */
#define CTF_V2_INFO_KIND(_info)   (((_info) & 0xf800) >> 11)
#define CTF_V2_INFO_ISROOT(_info) (((_info) & 0x0400) >> 10)
#define CTF_V2_INFO_VLEN(_info)   (((_info) & CTF_V2_MAX_VLEN))
#define CTF_V2_TYPE_INFO(_kind, _isroot, _vlen) \
        (((_kind) << 11) | (((_isroot) ? 1 : 0) << 10) | \
        ((_vlen) & CTF_V2_MAX_VLEN))

#define CTF_V3_MAX_VLEN           0x00ffffff
#define CTF_V3_INFO_KIND(info)    (((info) & 0xfc000000) >> 26)
#define CTF_V3_INFO_ISROOT(info)  (((info) & 0x02000000) >> 25)
#define CTF_V3_INFO_VLEN(info)    (((info) & CTF_V3_MAX_VLEN))

#define CTF_V3_TYPE_INFO(kind, isroot, vlen) \
        (((kind) << 26) | (((isroot) ? 1 : 0) << 25) | \
        ((vlen) & CTF_V3_MAX_VLEN))
```

The Label Section

When consuming **CTF** data, it is often useful to know whether two different **CTF** containers come from the same source base and version. For example, many kernel modules are built against a single collection of source code. A label is encoded into the **CTF** files that corresponds with the particular build. This ensures that if files on the system were to become mixed up from multiple releases, that they will not used together by tools, particularly when a child needs to refer to a type in the parent. Labels are used to prevent such a parent/child mismatch from occurring.

Each label is encoded in the file format using the following eight byte structure:

```
typedef struct ctf_lblent {
        uint32_t ctl_label;     /* ref to name of label */
        uint32_t ctl_typeidx;   /* last type associated with label */
} ctf_lblent_t;
```

Each label has two different components, a name and a type identifier. The name is encoded in the **ctl_label** field which is in the format defined in **String Identifiers**. Generally, the names of all labels are found in the internal string section.

The type identifier encoded in the **ctl_typeidx** field refers to the last type identifier that a label refers to in the current file. Labels only refer to types in the current file, if the **CTF** file is a child, then it will have the same label as its parent; however, its label will only refer to its types, not its parent's.

It is also possible, though rather uncommon, for a **CTF** file to have multiple labels. Labels are placed one after another, every eight bytes. When multiple labels are present, types may only belong to a single label.

The Object Section

The object section provides a mapping from **ELF** symbols of type **STT_OBJECT** in the symbol table, to a corresponding type identifier. These type identifiers are 16-bit unsigned integers in **CTF** version 2, and 32-bit unsigned integers in **CTF** version 3, encoded as described in **Type Identifiers**. If there is no information for an object, then the type identifier 0 is stored for that entry.

To interpret the object section requires access to the corresponding symbol table in the associated **ELF** object. Starting with the first symbol in the symbol table, each symbol is checked in turn to find the **STT_OBJECT** symbols that correspond to entries in the object section. Not every symbol found in the **ELF** symbol table is used. When searching the symbol table, a symbol is skipped if it matches any of the following conditions:

- The type is not **STT_OBJECT**.
- The section index is **SHN_UNDEF**.
- The name offset is 0, indicating no name.
- The section index is **SHN_ABS** and the value of the symbol is 0.
- The symbol name is **_START_** or **_END_**.

The entries in the object section are written in the same order as their corresponding symbols in the symbol table, so each identified symbol corresponds to the next available object section entry. The number of entries in the object section must correspond to the number of **STT_OBJECT** symbols found.

The Function Section

The function section of a **CTF** file encodes the types of both the function's arguments and the function's return value. This section consists of 16-bit unsigned integer values for **CTF** version 2, and of 32-bit unsigned integer values for **CTF** version 3. Similar to **The Object Section**, the function section encodes information for all symbols of type **STT_FUNCTION**, excepting those that fit specific criteria. Unlike with objects, because functions have a variable number of arguments, their entries start with a type encoding as defined in **Type Encoding**.

Functions which have no type information available are encoded as follows. In this case, the entry is complete, and nothing else is written. The next integer in the section starts the next entry:

```
CTF_V2_TYPE_INFO(CTF_K_UNKNOWN, 0, 0)  /* CTF version 2 */
CTF_V3_TYPE_INFO(CTF_K_UNKNOWN, 0, 0)  /* CTF version 3 */
```

Functions with type information are encoded as:

```
CTF_V2_TYPE_INFO(CTF_K_FUNCTION, 0, nargs)  /* CTF version 2 */
CTF_V3_TYPE_INFO(CTF_K_FUNCTION, 0, nargs)  /* CTF version 3 */
```

The encoding variable length (**vlen**) is used to convey the number of arguments to the function. If a function is a **varargs** type function, then the number of arguments is increased by one.

The next integer written provides the type identifier of the return type of the function, and is followed by a type identifier for each argument, if any exist, in the order that they appear in the function. When a function has a final varargs argument, it is encoded with the type identifier 0.

In the same manner as described in **The Object Section**, the entries in the function section match the order of the **STT_FUNCTION** symbols found in the symbol table. The rules for matching symbol to entry are similar, but slightly different than those for objects. While iterating over the symbol table, if any of the

following conditions are true, then the symbol is skipped and no corresponding entry is written:

- The type is not **STT_FUNCTION**.
- The section index is **SHN_UNDEF**.
- The name offset is 0, indicating no name.
- The symbol name is **_START_** or **_END_**.

The Type Section

The type section is the heart of the **CTF** data, providing information for the types used in the corresponding object. Each entry consists of a type structure, potentially followed by kind-specific variable data. There are two forms of type structure, a shorter one used for most purposes, and a longer one used for larger sizes. The variable data, when present, follows immediately after the type structure. The short and long form of type structure are defined as follows. The **CTF** version 2 and 3 versions of these structures differ in that version 2 uses 16-bit integers for some fields, while version 3 widens those fields. The version 2 and 3 variants of each structure are shown side by side, and differences are shown in boldface, to facilitate comparison.

```
#define CTF_V2_MAX_SIZE     0xfffe  /* max size of a type (bytes) */
#define CTF_V2_LSIZE_SENT    0xffff  /* sentinel for ctt_size */

#define CTF_V3_MAX_SIZE     0xfffffffe
#define CTF_V3_LSIZE_SENT    0xffffffff
```

```
typedef struct ctf_stype_v2 {
        uint32_t ctt_name;
        uint16_t ctt_info;
        union {
                uint16_t _size;
                uint16_t _type;
        } _u;
} ctf_stype_v2_t;

typedef struct ctf_type_v2 {
        uint32_t ctt_name;
        uint16_t ctt_info;
        union {
                uint16_t _size;
                uint16_t _type;
        } _u;
        uint32_t ctt_lsizehi;
        uint32_t ctt_lsizelo;
} ctf_type_v2_t;
```

```
struct ctf_stype_v3 {
        uint32_t ctt_name;
        uint32_t ctt_info;
        union {
                uint32_t _size;
                uint32_t _type;
        } _u;
} ctf_stype_v3;

typedef struct ctf_type_v3 {
        uint32_t ctt_name;
        uint32_t ctt_info;
        union {
                uint32_t _size;
                uint32_t _type;
        } _u;
        uint32_t ctt_lsizehi;
        uint32_t ctt_lsizelo;
} ctf_type_v3_t;
```

```
#define ctt_size _u._size    /* for types that have a size */
#define ctt_type _u._type    /* for types that ref. another type */
```

The long form is identical to the short, with the addition of 2 fields at the end used to encode large sizes. Due to their common layout, it is common for code that reads types to use a pointer to the large form to access the data, paying attention to the type kind and size to determine how to read the size, and the amount to increment the pointer between items.

Types are written out in order, with no padding in between them. The type ID for each type is implicit in its position within the type section. The first type entry has type ID 1, the second has type ID 2, and so forth. In a child object, the type ID has its **child bit** set, so in that case, the first identifier will have value 0x8000 for **CTF** version 2, or 0x80000000 for **CTF** version 3.

ctt_name is encoded as described in **String Identifiers**. The string that it points to is the name of the type. If the identifier points to an empty string (one that consists solely of a null terminator) then the type does not have a name. This is common with anonymous structures and unions that only have a typedef to identify them, as well as for pointers and qualifiers.

ctt_info, is encoded as described in **Type Encoding**. The type's kind tells us how to interpret the remaining data in the type structure, and any variable length (**vlen**) data that may exist.

Each type record has a **kind**. Some kinds convey size information, while others reference another type. The **ctt_size** and **ctt_type** fields are held in a union, so only one or the other can be used. The field used is determined by the kind. The kind-specific sections below describe the specific details.

The following kinds use **ctt_type**. **CTF_K_UNKNOWN** and **CTF_K_FORWARD** always set it to 0. The other types use it to hold a referenced type ID. The short type structure is always used for these types.

```
#define CTF_K_UNKNOWN   0
#define CTF_K_FORWARD   9

#define CTF_K_POINTER   3
#define CTF_K_FUNCTION  5
#define CTF_K_TYPEDEF   10
#define CTF_K_VOLATILE  11
#define CTF_K_CONST     12
#define CTF_K_RESTRICT  13
```

The following kinds convey sizes, and use **ctt_size**. The short type structure is used for sizes that fit within its limits, and the larger type structure is used otherwise.

```
#define CTF_K_INTEGER   1
#define CTF_K_FLOAT     2
#define CTF_K_ARRAY     4
#define CTF_K_STRUCT    6
#define CTF_K_UNION     7
#define CTF_K_ENUM      8
#define CTF_K_SLICE     14
```

Type sizes are measured in bytes. When the size to be represented will fit in **ctt_size**, the short form of the type structure is used. If the size is too large to fit, then a special sentinel value is written to **ctt_size** to indicate that fact, and the large form of the type structure is used. Although versions 2 and 3 use different structures and macros, the same approach applies to both.

CTF Version 2

CTF version 2 defines **ctt_size** as a **uint16_t** value. If the size to be represented is less than or equal to **CTF_V2_MAX_SIZE** (0xfffe), then the short type structure, **ctf_stype_v2** is used. Otherwise, the large form, **ctf_type_v2**, is used, **ctt_size** is set to a special sentinel value **CTF_V2_LSIZE_SENT** (0xffff), and the size is instead written to **ctt_lsizehi** and **ctt_lsizelo**.

```
ctf_type_v2 ctt;

if (size > CTF_V2_MAX_SIZE) {          /* 0xfffe */
        ctt.ctt_size = CTF_V2_LSIZE_SENT; /* 0xffff */
        ctt.ctt_lsizehi = CTF_SIZE_TO_LSIZE_HI(size);
        ctt.ctt_lsizelo = CTF_SIZE_TO_LSIZE_LO(size);
} else {
        ctt.ctt_size = size;
}
```

CTF Version 3

CTF version 3 defines **ctt_size** as a **uint32_t** value. If the size to be represented is less than or equal to **CTF_V3_MAX_SIZE** (0xfffffffe), then the short type structure, **ctf_stype_v3** is used. Otherwise, the large form, **ctf_type_v3**, is used, **ctt_size** is set to a special sentinel value **CTF_V3_LSIZE_SENT** (0xffffffff), and the size is instead written to **ctt_lsizehi** and **ctt_lsizelo**.

```
ctf_type_v3 ctt;

if (size > CTF_V3_MAX_SIZE) {          /* 0xfffffffe */
        ctt.ctt_size = CTF_V3_LSIZE_SENT; /* 0xffffffff */
        ctt.ctt_lsizehi = CTF_SIZE_TO_LSIZE_HI(size);
        ctt.ctt_lsizelo = CTF_SIZE_TO_LSIZE_LO(size);
} else {
        ctt.ctt_size = size;
}
```

Encoding of Integer Types (CTF_K_INTEGER)

Integers, which are of type **CTF_K_INTEGER**, have no variable length arguments, and should specify a **vlen** of 0. The type structure is followed by a 32-bit unsigned integer which describes the encoding. The **ctt_size** field describes the size of the integer, in bytes. In general, integer sizes will be rounded up to the closest power of two.

The integer encoding contains three different pieces of information:

- The encoding of the integer.
- The offset in **bits** of the type.
- The size in **bits** of the type.

This encoding can be expressed through the following macros:

```
#define CTF_INT_ENCODING(data)  (((data) & 0xff000000) >> 24)
#define CTF_INT_OFFSET(data)    (((data) & 0x00ff0000) >> 16)
#define CTF_INT_BITS(data)      (((data) & 0x0000ffff))

#define CTF_INT_DATA(encoding, offset, bits) \
    (((encoding) << 24) | ((offset) << 16) | (bits))
```

The following flags are defined for the encoding:

```
#define CTF_INT_SIGNED        0x01
#define CTF_INT_CHAR          0x02
#define CTF_INT_BOOL          0x04
#define CTF_INT_VARARGS       0x08
```

By default, an integer is considered to be unsigned, unless the **CTF_INT_SIGNED** flag set. The flag **CTF_INT_CHAR** indicates that the integer is of a type that stores character data. For example, **CTF_INT_CHAR** is set for the intrinsic C type **char**. **CTF_INT_BOOL** indicates that the integer represents a boolean type. For example, **CTF_INT_BOOL** is set for the intrinsic C type **_Bool**. **CTF_INT_VARARGS** indicates that the integer is used as part of a variable number of arguments. This encoding is rather uncommon.

The offset and size of a **CTF_K_INTEGER** can be overridden by a preceding **CTF_K_SLICE** type. See **Encoding of Slice Types (CTF_K_SLICE)**.

A **CTF_K_INTEGER** type with both the offset and size set to 0 represents the C language **void** type. The **CTF_K_SLICE** type cannot be applied to this special case of **CTF_K_INTEGER**.

Encoding of Float Types (CTF_K_FLOAT)

Floats, which are of type **CTF_K_FLOAT**, are similar to their integer counterparts. They have no variable length arguments, specify a **vlen** of 0, and the type structure is followed by a 32-bit unsigned integer which describes the kind of float. The **ctt_size** field provides the size of the float, in bytes. The float encoding provides three different pieces of information:

- The specific kind of float.
- The offset in **bits** of the float.
- The size in **bits** of the float.

This encoding can be expressed through the following macros:

```
#define CTF_FP_ENCODING(data)   (((data) & 0xff000000) >> 24)
#define CTF_FP_OFFSET(data)     (((data) & 0x00ff0000) >> 16)
#define CTF_FP_BITS(data)       (((data) & 0x0000ffff))

#define CTF_FP_DATA(encoding, offset, bits) \
    (((encoding) << 24) | ((offset) << 16) | (bits))
```

Unlike **CTF_K_INTEGER**, which uses flags to describe various attributes, **CTF_K_FLOAT** uses a simple integer to identify the floating format, each of which fully describes all attributes of a specific floating format.

```
#define CTF_FP_SINGLE   1       /* IEEE 32-bit float */
#define CTF_FP_DOUBLE   2       /* IEEE 64-bit float */
#define CTF_FP_CPLX     3       /* Complex */
#define CTF_FP_DCPLX    4       /* Double complex */
#define CTF_FP_LDCPLX   5       /* Long double complex */
#define CTF_FP_LDOUBLE  6       /* Long double */
#define CTF_FP_INTRVL   7       /* Interval (2x32-bit) */
#define CTF_FP_DINTRVL  8       /* Double interval (2x64-bit) */
#define CTF_FP_LDINTRVL 9       /* Long double interval (2x128-bit) */
#define CTF_FP_IMAGRY   10      /* Imaginary (32-bit) */
#define CTF_FP_DIMAGRY  11      /* Long imaginary (64-bit) */
#define CTF_FP_LDIMAGRY 12      /* Long double imaginary (128-bit) */
```

Encoding of Array Types (CTF_K_ARRAY)

Arrays, which are of type **CTF_K_ARRAY**, have no variable list entries, and therefore set **vlen** to 0. The type structure is followed by a structure which describes the number of elements in the array (**cta_nelems**), the type identifier of the elements in the array (**cta_contents**), and the type identifier of the index of the array (**cta_index**). With arrays, **ctt_size** is set to 0.

The **CTF** version 2 and 3 versions of this structure differ in that version 2 uses 16-bit integers for some fields, while version 3 widens those fields. The version 2 and 3 variants of each structure are shown side by side, and the differences are shown in boldface, to facilitate comparison.

```
typedef struct ctf_array_v2 {
        uint16_t cta_contents;
        uint16_t cta_index;
        uint32_t cta_nelems;
} ctf_array_v2_t;
```

```
typedef struct ctf_array_v3 {
        uint32_t cta_contents;
        uint32_t cta_index;
        uint32_t cta_nelems;
} ctf_array_v3_t;
```

cta_contents and **cta_index** are type identifiers, encoded as described in **Type Identifiers**. **cta_nelems** is an unsigned count of the number of elements. This count will be 0 when describing C99 flexible array members.

Encoding of Function Types (CTF_K_FUNCTION)

Function types, which are of kind **CTF_K_FUNCTION**, use the variable length (**vlen**) to provide the number of arguments in the function. When the function has a final argument which is a varargs, then the argument count is incremented by one to account for the variable argument. The **ctt_type** field is used to hold the type identifier of the function return type.

The variable data holds a list of type identifiers for the arguments of the function, if any. For **CTF** version 2, each argument is represented by a 16-bit **uint16_t** value, while for **CTF** version 3, each argument is a 32-bit **uint32_t** value. Each is encoded as described in **Type Identifiers**. If the function's last argument is of type varargs, it is written using type identifier 0.

In **CTF** version 2, an extra type identifier with value 0 is added following the final argument, if needed to

maintain four-byte alignment for the data that follows. If present, this pad value is not included in the argument count. In **CTF** version 3, four-byte alignment occurs naturally and no padding is used.

Encoding of Structure (CTF_K_STRUCT) and Union (CTF_K_UNION) Types

Structures and Unions, which are encoded with **CTF_K_STRUCT** and **CTF_K_UNION** respectively, are very similar constructs in C. As such, their encoding in **CTF** is also very similar. The main difference between structures and unions is that members of a structure are laid out in memory, in order, one after the other, while in a union, all members share the same memory.

The variable length (**vlen**) for structures and unions specifies the number of members. The value of **ctt_size** is set to the size of the structure or union. As with the type structure, the structure used to encode members come in two forms, a smaller one one that serves most uses, and a large form used for larger sizes. The overall size of the struct or union determines which form is used to encode members in the variable list. The following definitions describe these two member structures. The **CTF** version 2 and 3 versions of these structures differ in that version 2 uses 16-bit integers for some fields, while version 3 widens those fields. The version 2 and 3 variants of each structure are shown side by side, and differences are shown in boldface, to facilitate comparison.

```
#define CTF_V2_LSTRUCT_THRESH   (1 << 13)   /* 8192 */
#define CTF_V3_LSTRUCT_THRESH   (1 << 29)   /* 536870912 */

#define CTF_LMEM_OFFSET(_ctlmp) \
        (((uint64_t)(_ctlmp)->ctlm_offsethi) << 32 | \
        (_ctlmp)->ctlm_offsetlo)
#define CTF_OFFSET_TO_LMEMHI(_offset)   \
        ((uint32_t)((uint64_t)(_offset) >> 32))
#define CTF_OFFSET_TO_LMEMLO(_offset)   ((uint32_t)(_offset))
```

```
typedef struct ctf_member_v2 {
        uint32_t ctm_name;
        uint16_t ctm_type;
        uint16_t ctm_offset;
} ctf_member_v2_t;

typedef struct ctf_lmember_v2 {
        uint32_t ctlm_name;
        uint16_t ctlm_type;
        uint16_t ctlm_pad;
        uint32_t ctlm_offsethi;
        uint32_t ctlm_offsetlo;
} ctf_lmember_v2_t;
```

```
typedef struct ctf_member_v3 {
        uint32_t ctm_name;
        uint32_t ctm_type;
        uint32_t ctm_offset;
ctf_member_v3_t;

typedef struct ctf_lmember_v3 {
        uint32_t ctlm_name;
        uint32_t ctlm_type;

        uint32_t ctlm_offsethi;
        uint32_t ctlm_offsetlo;
ctf_lmember_v3_t;
```

When the size of a structure or union is greater than or equal to the large member threshold defined for the version of CTF (**CTF_V*_LSTRUCT_THRESH**), then the large **lmember** structure is used to encode members, rather than the smaller **member** structure. All members are encoded using the same structure.

Both **ctm_name** and **ctlm_name** refer to the name of the member. The name is encoded as an offset into the string table as described in **String Identifiers**. The members **ctm_type** and **ctlm_type** both refer to the type of the member. They are encoded as described in **Type Identifiers**.

The last piece of information that is present is the offset which describes the offset in memory at which the member begins. For unions, this value will always be 0 because each member of a union has an offset of 0. For structures, this is the offset in **bits** at which the member begins. Note that a compiler may lay out a type with padding. This means that the difference in offset between two consecutive members may be larger than the size of the member. When the size of the overall structure is strictly less than the large member threshold, the smaller **member** structure is used, and the offset in bits is stored in the member **ctm_offset**. However, when the size of the structure exceeds the large member threshold, the larger **member** structure is used, and the number of bits is split into two 32-bit quantities. One member, **ctlm_offsethi**, represents the upper 32 bits of the offset, while the other member, **ctlm_offsetlo**, represents the lower 32 bits of the offset. These can be joined together to get a 64-bit sized offset in bits using the **CTF_LMEM_OFFSET** macro shown above.

Encoding of Enumeration Types (CTF_K_ENUM)

Enumerations, which are of kind **CTF_K_ENUM**, map integer values to symbolic names. The mappings are referred to as **enumerators**. Enumerations use the variable length (**vlen**) to specify the number of enumerators. In C, an enumeration is always equivalent to the intrinsic type **int**, thus the value of **ctt_size** will be the size of an **int**, which on Solaris systems is always 4.

Each enumerator is described by the following structure in the variable list:

```
typedef struct ctf_enum {
        uint32_t cte_name;  /* reference to name in strtab */
        int32_t cte_value;  /* value associated with this name */
} ctf_enum_t;
```

cte_name refers to the name of the enumerator's value, and is encoded according to the rules described in **String Identifiers**. The **cte_value** field provides the integer value of this enumerator.

Unlike **CTF_K_INTEGER**, **CTF_K_ENUM** does not specify an encoding, as the encoding would always be that of a C **int**. However, the offset and size of a **CTF_K_ENUM** can be overridden by a preceding **CTF_K_SLICE** type. See **Encoding of Slice Types (CTF_K_SLICE)**.

Encoding of Forward References (CTF_K_FORWARD)

Forward references, which are of kind **CTF_K_FORWARD**, refer to types which may not have a definition at all, only a name. If the **CTF** file is a child, then it may be that the forward is resolved to an actual type in the parent, otherwise the definition may be in another **CTF** container, or may not be known at all. The only field of the type structure that is used for a forward declaration is **ctt_name**, which points to the name of the forward reference in the string table. The **ctt_type** field should be set to 0. This type has no variable list entries, and **vlen** is set to 0. There is no other information recorded for forward references.

Encoding of Pointers, Typedefs, Volatile, Const, and Restrict

Pointers, typedefs, volatile, const, and restrict all refer to another type. In the case of typedefs, they provide an alternate name, while volatile, const, and restrict change how the type is interpreted in the C programming language. This covers the **CTF** kinds **CTF_K_POINTER**, **CTF_K_TYPEDEF**, **CTF_K_VOLATILE**, **CTF_K_RESTRICT**, and **CTF_K_CONST**. These types have no variable list entries, and **vlen** is set to 0. The **ctt_type** field is used to refer to the modified base type.

Encoding of Slice Types (CTF_K_SLICE)

Slices, which are of kind **CTF_K_SLICE**, override the offset and width of a referenced integer (**CTF_K_INTEGER**) or enumeration (**CTF_K_ENUM**) type. Slices are used to represent bitfields in structure or union types, when the width of the field differs from that of the underlying referenced type.

Slices are nameless, and have no variable list entries, so **ctt_name**, and **vlen** are set to 0. The type structure is followed by a structure which specifies the referenced type, and the overriding offset and size to be

applied to that type. For slices, **ctt_size** is set to the number of bytes required to represent the size given by **cts_bits**, rounded up to the nearest power of 2.

The **CTF** version 2 and 3 versions of this structure differ in that version 2 uses 16-bit integers for some fields, while version 3 widens those fields. The version 2 and 3 variants of each structure are shown side by side, and the differences are shown in boldface, to facilitate comparison.

```
typedef struct ctf_slice_v2 {
        uint16_t cts_type;
        uint16_t cts_offset;
        uint16_t cts_bits;
        uint16_t cts_pad;
} ctf_slice_v2_t;
```

```
typedef struct ctf_slice_v3 {
        uint32_t cts_type;
        uint16_t cts_offset;
        uint16_t cts_bits;

} ctf_slice_v3_t;
```

cts_type refers to the type modified by the slice. The **cts_offset** and **cts_bits** fields provides the offset and size of the modified type, specified in **bits**.

Encoding of Unknown Types (CTF_K_UNKNOWN)

Types with the kind **CTF_K_UNKNOWN** are used to indicate gaps in the type identifier space. Such entries consume an identifier, but do not define anything. Nothing should refer to these gap identifiers. The **ctt_name** and **ctt_type** fields should be set to 0. This type has no variable list entries, and **vlen** is set to 0.

Dependencies Between Types

C types can be imagined as a directed, cyclic, graph. Structures and unions may refer to each other in a way that creates a cyclic dependency. In cases such as these, the entire type section must be read in and processed. Consumers must not assume that every type can be laid out in dependency order; they cannot.

The String Section

The string section is the final section of a **CTF** file. This section contains the strings that are referenced throughout the other sections. **CTF** string tables are modeled after **ELF** string table sections, and follow the same rules.

The string table is an array of 8-bit character bytes. Each string is written to the table in turn, including a NULL termination. The order of the strings within the table, relative to each other, is unspecified, and should not be relied on. All references to the string table are made by specifying the offset of its first character. The first byte in the string table is always a null termination, so offset 0 always represents an empty string.

Generally, all characters in the string table come from the 7-bit ASCII character set, as most C compilers limit the characters used in identifiers to this range. However, any extended characters sets should be written as UTF-8.

Data Encoding and ELF Considerations

CTF data is generally included in **ELF** objects. The **ELF** header specifies information that identifies the machine architecture, and byte order, for the file. A **CTF** container inside such an object must be written using the same byte order as that of the ELF object.

Other than byte order, **CTF** is a machine independent format, and there should be no other differences between architectures. Where this document refer to non-fixed size C integral types, definitions that match those of the **ILP32**, and **LP64** models should be assumed.

When placing a **CTF** container within an **ELF** object, there are conventions that must be followed in order for that **CTF** data to be usable. In particular, a given **ELF** object should only contain a single **CTF** section. Multiple containers should be merged together into a single one. See **ctfmerge**(1).

The **CTF** file should be included in its own **ELF** section. The section's name must be **.SUNW_ctf**. The type of the section should be **SHT_SUNW_CTF**, although for compatibility with historical use, **SHT_PROGBITS** is also allowed. The section header for the **ELF** section containing the **CTF** data should link to the symbol table (**sh_link**), and specify an address alignment of 4 (**sh_addralign**).

The symbol table associated to the **.SUNW_ctf** section by the **sh_link** section header field must be one of the three supported symbol tables, **.symtab** (**SHT_SYMTAB**), **.dynsym** (**SHT_DYNSYM**), or **.SUNW_ldynsym** (**SHT_SUNW_LDYNSYM**). When the **.SUNW_ldynsym** is specified, the actual association is to the concatenation of the **.SUNW_ldynsym** and **.dynsym** symbol tables, treated as a single logical table.

HISTORY

The **CTF** version 2 format has been used in the construction of Solaris since Sun Solaris 9. Version 1, the initial development version, was never included in a user visible release.

The required **ELF** section type for **.SUNW_ctf** sections was changed from **SHT_PROGBITS** to **SHT_SUNW_CTF** in the Oracle Solaris 11.4.75 release. Support for existing objects with **CTF** in **SHT_PROGBITS** sections is retained to to support historical usage.

Support for **CTF** version 3, was added in the Oracle Solaris 11.4.81 release.

SEE ALSO

ctfconvert(1), **ctfdump**(1), **ctfmerge**(1), **ld**(1), **mdb**(1), **libz**(3), **dtrace**(8)