NAME

ctf - Compact C Type Format

SYNOPSIS

#include <sys/ctf.h>

DESCRIPTION

ctf 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 and is tagged as **SHT_PROGBITS** 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 a **ctf** file is often embedded inside a file, rather than being a standalone file itself, it may also be referred to as a **ctf container**.

On FreeBSD systems, **ctf** data is consumed by dtrace(1). Programmatic access to **ctf** data can be obtained through libctf.

The **ctf** file format is broken down into seven different sections. The first two sections are the **preamble** and **header**, which describe the version of the **ctf** file, the links it has to other **ctf** files, and the sizes of the other sections. The next section 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 themselves, and finally the last section is the **string** section, which contains the names of types, enumerations, members, and labels.

While strictly speaking, only the **preamble** and **header** are required, to be actually useful, both the type and string sections are necessary.

A **ctf** file may contain all of the type information that it requires, or it may optionally refer to another **ctf** file which holds 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 one 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 kernel modules in illumos are uniquified against the kernel module **genunix** and the type information that comes from the **IP** module. This means that a module only has types that are unique to itself and the most common types in the kernel are not duplicated. Uniquification is not used when building kernel modules on FreeBSD.

FILE FORMAT

This documents version *three* of the **ctf** file format. The ctfconvert(1) and ctfmerge(1) utilities emit **ctf** version 3, and all other applications and libraries can operate on versions 2 and 3.

The file format can be summarized with the following image, the following sections will cover this in more detail.

```
          +-------------+  0t0
+--------| Preamble    |
|        +-------------+  0t4
|+-------| Header      |
||       +-------------+  0t36 + cth_lbloff
||+------| Labels      |
|||      +-------------+  0t36 + cth_objtoff
|||+-----| Objects     |
||||     +-------------+  0t36 + cth_funcoff
||||+----| Functions   |
|||||    +-------------+  0t36 + cth_typeoff
|||||+---| Types       |
||||||   +-------------+  0t36 + cth_stroff
||||||+--| Strings     |
|||||||  +-------------+  0t36 + cth_stroff + cth_strlen
|||||||
|||||||
|||||||
|||||||    +-- magic -   vers   flags
|||||||    |          |    |      |
|||||||    +------+------+------+------+
+---------| 0xcf | 0xf1 | 0x03 | 0x00 |
 ||||||    +------+------+------+------+
 ||||||    0      1      2      3      4
 ||||||
 ||||||    + parent label         + objects
 ||||||    |      + parent name |      + functions    + strings
 ||||||    |      |      + label |      |      + types |       + strlen
```

```
|||||    |       |     |       |     |      |       |       |
|||||   +-------+------+------+------+------+-------+-------+-------+
+--------| 0x00 | 0x00 | 0x00 | 0x08 | 0x36 | 0x110 | 0x5f4 | 0x611 |
 |||||   +------+------+------+------+------+-------+-------+-------+
 |||||   0x04   0x08   0x0c   0x10   0x14    0x18    0x1c    0x20   0x24
 |||||
 |||||         + Label name
 |||||         |       + Label type
 |||||         |       |       + Next label
 |||||         |       |       |
 |||||       +-------+------+-----+
 +-----------| 0x01  | 0x42 | ... |
  ||||       +-------+------+-----+
  ||||  cth_lbloff   +0x4   +0x8  cth_objtoff
  ||||
  ||||
  |||| Symidx  0t15   0t43   0t44
  ||||       +------+------+------+-----+
  +----------| 0x00 | 0x42 | 0x36 | ... |
   |||       +------+------+------+-----+
   ||| cth_objtoff  +0x4   +0x8   +0xc   cth_funcoff
   |||
   |||        + CTF_TYPE_INFO         + CTF_TYPE_INFO
   |||        |        + Return type  |
   |||        |        |       + arg0 |
   |||       +--------+------+------+-----+
   +---------| 0x2c10 | 0x08 | 0x0c | ... |
    ||       +--------+------+------+-----+
    || cth_funcff     +0x4   +0x8   +0xc  cth_typeoff
    ||
    ||         + ctf_stype_t for type 1
    ||         |  integer           + integer encoding
    ||         |                    |          + ctf_stype_t for type 2
    ||         |                    |          |
    ||       +--------------------+-----------+-----+
    +--------| 0x19 * 0xc01 * 0x0 | 0x1000000 | ... |
     |       +--------------------+-----------+-----+
     | cth_typeoff               +0x0c      +0x10  cth_stroff
     |
     |     +--- str 0
     |    |   +--- str 1       + str 2
     |    |   |               |
     |    v   v               v
     |   +----+---+---+---+----+---+---+---+---+---+---+----+
     +---| \0 | i | n | t | \0 | f | o | o | _ | t | \0 |
         +----+---+---+---+----+---+---+---+---+---+---+----+
          0    1   2   3   4    5   6   7   8   9   10   11
```

Every **ctf** file begins with a **preamble**, followed by a **header**. The **preamble** is defined as follows:

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

The **preamble** is four bytes long and must be four byte aligned. This **preamble** defines the version of the **ctf** file which defines the format of the rest of the header. While the header may change in subsequent versions, the preamble will not change across versions, though the interpretation of its flags may change from version to version. The *ctp_magic* member defines the magic number for the **ctf** file format. This must always be 0xcff1. If another value is encountered, then the file should not be treated as a **ctf** file. The *ctp_version* member defines the version of the **ctf** file. 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. Finally, the *ctp_flags* member describes aspects of the file which modify its interpretation. The following flags are currently defined:

```
#define    CTF_F_COMPRESS              0x01
```

The flag **CTF_F_COMPRESS** indicates that the body of the **ctf** file, all the data following the **header**, has been compressed through the **zlib** library and its **deflate** algorithm. 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 **header**, always refer to the **uncompressed** data.

In versions two and three of the **ctf** file format, the **header** denotes whether or not this **ctf** file is the child of another **ctf** file and also indicates the size of the remaining sections. The structure for the **header** logically contains a copy of the **preamble** and the two have a combined size of 36 bytes.

```
typedef struct ctf_header {
        ctf_preamble_t cth_preamble;
        uint32_t cth_parlabel;      /* ref to name of 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 members *cth_parlabel* and *cth_parname*, are used to identify the parent. The value of both members 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 the subsection on *String Identifiers*. If the value of either is zero, then there is no entry for that member. If the member *cth_parlabel* is set, then the *ctf_parname* member 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 members (excepting *cth_strlen*) describe the beginning of the corresponding sections. These offsets are relative to the end of the **header**. Therefore, something with an offset of 0 is at an offset of thirty-six bytes relative to the start of the **ctf** file. The difference between members indicates the size of the section itself. Different offsets have different alignment requirements. The start of the *cth_objtoff* and *cth_funcoff* must be two byte aligned, while the sections *cth_lbloff* and *cth_typeoff* must be four-byte aligned. The section *cth_stroff* has no alignment requirements. To calculate the size of a given section, excepting the **string** section, one should subtract the offset of the section from the following one. For example, the size of the **types** section can be calculated by subtracting *cth_typeoff* from *cth_stroff*.

Finally, the member *cth_strlen* describes the length of the string section itself. From it, you can also calculate the size of the entire **ctf** file by adding together the size of the **ctf_header_t**, the offset of the string section in *cth_stroff*, and the size of the string section in *cth_srlen*.

Type Identifiers

Through the **ctf** data, types are referred to by identifiers. A given **ctf** file supports up to 2147483646 (0x7ffffffe) types. **ctf** version 2 had a much smaller limit of 32767 types. The first valid type identifier is 0x1. When a given **ctf** file is a child, indicated by a non-zero entry for the **header**'s *cth_parname*, then the first valid type identifier is 0x80000000 and the last is 0xfffffffe. In this case, type identifiers 0x1 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.

The type identifier zero is a sentinel value used to indicate that there is no type information available or it is an unknown type.

Throughout the file format, the identifier is stored in different sized values; however, the minimum size to represent a given identifier is a **uint16_t**. Other consumers of **ctf** information may use larger or opaque identifiers.

String Identifiers

String identifiers are always encoded as four byte unsigned integers which are an offset into a string table. The **ctf** format supports two different string tables which have an identifier of zero or one. This identifier is stored in the high-order bit of the unsigned four byte offset. Therefore, the maximum supported offset into one of these tables is 0x7ffffffff.

Table identifier zero, always refers to the **string** section in the CTF file itself. String table identifier one refers to an external string table which is the ELF string table for the ELF symbol table associated with the **ctf** container.

Type Encoding

Every **ctf** type begins with metadata encoded into a **uint32_t**. This encoded information tells us three different pieces of information:

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

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.

The current version of the file format defines 14 different kinds. The interpretation of these different kinds will be discussed in the section *The Type Section*. If a kind is encountered that is not listed below, then it is not a valid **ctf** file. The kinds are defined as follows:

```
#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_TYPEDEF10
    #define   CTF_K_VOLATILE          11
    #define   CTF_K_CONST    12
    #define   CTF_K_RESTRICT          13
```

Programs directly reference many types; however, other types are referenced indirectly because they are part of some other structure. These 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 bit 25 set.

The variable length section is specific to each kind and is discussed in the section *The Type Section*.

The following macros are useful for constructing and deconstructing the encoded type information:

```
    #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, when building illumos, there are many kernel modules that 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 are not used together by tools, particularly when a child needs to refer to a type in the parent. Because they are linked using the type identifiers, if the wrong parent is used then the wrong type will be encountered. Note that this mechanism is not currently used on FreeBSD. In particular, kernel modules built on FreeBSD each contain a complete type graph.

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 this label */
} ctf_lblent_t;
```

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

The type identifier encoded in the member *ctl_typeidx* 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 type identifier. Every entry in this section is a **uint32_t** which contains a type identifier as described in the section *Type Identifiers*. If there is no information for an object, then the type identifier 0x0 is stored for that entry.

To walk the object section, you need to have a corresponding **symbol table** in the ELF object that contains the **ctf** data. Not every object is included in this section. Specifically, when walking the symbol table, an entry is skipped if it matches any of the following conditions:

- The symbol type is not **STT_OBJECT**
- The symbol's section index is **SHN_UNDEF**
- The symbol's name offset is zero
- The symbol's section index is **SHN_ABS** and the value of the symbol is zero.
- The symbol's name is _START_ or _END_. These are skipped because they are used for scoping local symbols in ELF.

The following sample code shows an example of iterating the object section and skipping the correct symbols:

```
#include <gelf.h>
#include <stdio.h>

/*
 * Given the start of the object section in a CTFv3 file, the number of symbols,
 * and the ELF Data sections for the symbol table and the string table, this
 * prints the type identifiers that correspond to objects. Note, a more robust
 * implementation should ensure that they don't walk beyond the end of the CTF
 * object section.
 *
 * An implementation that handles CTFv2 must take into account the fact that
 * type identifiers are 16 bits wide rather than 32 bits wide.
 */
static int
walk_symbols(uint32_t *objtoff, Elf_Data *symdata, Elf_Data *strdata,
    long nsyms)
{
        long i;
        uintptr_t strbase = strdata->d_buf;

        for (i = 1; i < nsyms; i++, objftoff++) {
                const char *name;
                GElf_Sym sym;

                if (gelf_getsym(symdata, i, &sym) == NULL)
                        return (1);

                if (GELF_ST_TYPE(sym.st_info) != STT_OBJECT)
                        continue;
                if (sym.st_shndx == SHN_UNDEF || sym.st_name == 0)
                        continue;
                if (sym.st_shndx == SHN_ABS && sym.st_value == 0)
                        continue;
                name = (const char *)(strbase + sym.st_name);
                if (strcmp(name, "_START_") == 0 || strcmp(name, "_END_") == 0)
                        continue;

                (void) printf("Symbol %d has type %d0, i, *objtoff);
        }

        return (0);
}
```

The Function Section

The function section of the **ctf** file encodes the types of both the function’s arguments and the function’s return value. 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, they start with a type encoding as defined in *Type Encoding*, which is the size of a **uint32_t**. For functions which have no type information available, they are encoded as CTF_V3_TYPE_INFO(CTF_K_UNKNOWN, 0, 0). Functions with arguments are encoded differently. Here, the variable length is turned into the number of arguments in the function. If a function is a **varargs** type function, then the number of arguments is increased by one. Functions with type information are encoded as: CTF_V3_TYPE_INFO(CTF_K_FUNCTION, 0, nargs).

For functions that have no type information, nothing else is encoded, and the next function is encoded. For functions with type information, the next **uint32_t** is encoded with the type identifier of the return type of the function. It is followed by each of the type identifiers of the arguments, if any exist, in the order that they appear in the function. Therefore, argument 0 is the first type identifier and so on. When a function has a final varargs argument, that is encoded with the type identifier of zero.

Like *The Object Section*, the function section is encoded in the order of the symbol table. It has similar, but slightly different considerations from objects. While iterating the symbol table, if any of the following conditions are true, then the entry is skipped and no corresponding entry is written:

- The symbol type is not **STT_FUNCTION**
- The symbol's section index is **SHN_UNDEF**
- The symbol's name offset is zero
- The symbol's name is _START_ or _END_. These are skipped because they are used for scoping local symbols in ELF.

The Type Section

The type section is the heart of the **ctf** data. It encodes all of the information about the types themselves. The base of the type information comes in two forms, a short form and a long form, each of which may be followed by a variable number of arguments. The following definitions describe the short and long forms:

```
#define   CTF_V3_MAX_SIZE               0xfffffffe /* max size of a type in bytes */
#define   CTF_V3_LSIZE_SENT      0xffffffff /* sentinel for ctt_size */
#define   CTF_V3_MAX_LSIZE       UINT64_MAX

struct ctf_stype_v3 {
          uint32_t ctt_name;  /* reference to name in string table */
          uint32_t ctt_info;    /* encoded kind, variant length */
          union {
                    uint32_t _size;      /* size of entire type in bytes */
                    uint32_t _type;      /* reference to another type */
          } _u;
};
```

```
struct ctf_type_v3 {
        uint32_t ctt_name;  /* reference to name in string table */
        uint32_t ctt_info;    /* encoded kind, variant length */
        union {
                uint32_t _size;        /* always CTF_LSIZE_SENT */
                uint32_t _type; /* do not use */
        } _u;
        uint32_t ctt_lsizehi;/* high 32 bits of type size in bytes */
        uint32_t ctt_lsizelo;/* low 32 bits of type size in bytes */
};

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

Type sizes are stored in **bytes**. The basic small form uses a **uint32_t** to store the number of bytes. If the number of bytes in a structure would exceed 0xfffffffe, then the alternate form, the **struct ctf_type_v3**, is used instead. To indicate that the larger form is being used, the member *ctt_size* is set to value of **CTF_V3_LSIZE_SENT** (0xffffffff). In general, when going through the type section, consumers use the **struct ctf_type_v3** structure, but pay attention to the value of the member *ctt_size* to determine whether they should increment their scan by the size of **struct ctf_stype_v3** or **struct ctf_type_v3**. Not all kinds of types use **ctt_size**. Those which do not, will always use the **struct ctf_stype_v3** structure. The individual sections for each kind have more information.

Types are written out in order. Therefore the first entry encountered has a type id of 0x1, or 0x8000 if a child. The member *ctt_name* is encoded as described in the section *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 name them, as well as pointers and qualifiers.

The next member, the *ctt_info*, is encoded as described in the section *Type Encoding*. The type's kind tells us how to interpret the remaining data in the **struct ctf_type_v3** and any variable length data that may exist. The rest of this section will be broken down into the interpretation of the various kinds.

Encoding of Integers

Integers, which are of type **CTF_K_INTEGER**, have no variable length arguments. Instead, they are followed by a **uint32_t** which describes their encoding. All integers must be encoded with a variable length of zero. The *ctt_size* member describes the length 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 at this time:

```
#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 it has the **CTF_INT_SIGNED** flag set. If the flag **CTF_INT_CHAR** is set, that indicates that the integer is of a type that stores character data, for example the intrinsic C type **char** would have the **CTF_INT_CHAR** flag set. If the flag **CTF_INT_BOOL** is set, that indicates that the integer represents a boolean type. For example, the intrinsic C type **_Bool** would have the **CTF_INT_BOOL** flag set. Finally, the flag **CTF_INT_VARARGS** indicates that the integer is used as part of a variable number of arguments. This encoding is rather uncommon.

Encoding of Floats

Floats, which are of type **CTF_K_FLOAT**, are similar to their integer counterparts. They have no variable length arguments and are followed by a four byte encoding which describes the kind of float that exists. The *ctt_size* member is the size, in bytes, of the float. The float encoding has three different pieces of information inside of it:

- The specific kind of float that exists
- 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))
```

Where as the encoding for integers is a series of flags, the encoding for floats maps to a specific kind of float. It is not a flag-based value. The kinds of floats correspond to both their size, and the encoding. This covers all of the basic C intrinsic floating point types. The following are the different kinds of floats represented in the encoding:

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

Encoding of Arrays

Arrays, which are of type **CTF_K_ARRAY**, have no variable length arguments. They are followed by a structure which describes the number of elements in the array, the type identifier of the elements in the array, and the type identifier of the index of the array. With arrays, the *ctt_size* member is set to zero. The structure that follows an array is defined as:

```
struct ctf_array_v3 {
        uint32_t cta_contents;          /* reference to type of array contents */
        uint32_t cta_index; /* reference to type of array index */
        uint32_t cta_nelems;            /* number of elements */
};
```

The *cta_contents* and *cta_index* members of the **struct ctf_array_v3** are type identifiers which are encoded as per the section *Type Identifiers*. The member *cta_nelems* is a simple four byte unsigned count of the number of elements. This count may be zero when encountering C99's flexible array members.

Encoding of Functions

Function types, which are of type **CTF_K_FUNCTION**, use the variable length list to be the number of arguments in the function. When the function has a final member which is a varargs, then the argument count is incremented by one to account for the variable argument. Here, the *ctt_type* member is encoded with the type identifier of the return type of the function. Note that the *ctt_size* member is not used here.

The variable argument list contains the type identifiers for the arguments of the function, if any. Each one is represented by a **uint32_t** and encoded according to the *Type Identifiers* section. If the function's last argument is of type varargs, then it is also written out, but the type identifier is zero. This is included in the count of the function's arguments. In **ctf** version 2, an extra type identifier may follow the argument and return type identifiers in order to maintain four-byte alignment for the following type definition. Such a type identifier is not included in the argument count and has a value of zero. In **ctf** version 3, four-byte alignment occurs naturally and no padding is used.

Encoding of Structures and Unions

Structures and Unions, which are encoded with **CTF_K_STRUCT** and **CTF_K_UNION** respectively, are very similar constructs in C. The main difference between them is the fact that members of a structure follow one another, where as in a union, all members share the same memory. They are also very similar in terms of their encoding in **ctf**. The variable length argument for structures and unions represents the number of members that they have. The value of the member *ctt_size* is the size of the structure and union. There are two different structures which are used to encode members in the variable list. When the size of a structure or union is greater than or equal to the large member threshold, 536870912, then a different structure is used to encode the member; all members are encoded using the same structure. The structure for members is as follows:

```
struct ctf_member_v3 {
        uint32_t ctm_name;      /* reference to name in string table */
        uint32_t ctm_type;  /* reference to type of member */
        uint32_t ctm_offset;    /* offset of this member in bits */
};

struct ctf_lmember_v3 {
        uint32_t ctlm_name;     /* reference to name in string table */
        uint32_t ctlm_type; /* reference to type of member */
        uint32_t ctlm_offsethi; /* high 32 bits of member offset in bits */
        uint32_t ctlm_offsetlo; /* low 32 bits of member offset in bits */
};
```

Both the *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 by the section *String Identifiers*. The members **ctm_type** and **ctlm_type** both refer to the type of the member. They are encoded as per the section *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 zero because each member of a union has an offset of zero. 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 536870912 bytes, the normal structure, **struct ctf_member_v3**, is used and the offset in bits is stored in the member *ctm_offset*. However, when the size of the structure is greater than or equal to 536870912 bytes, then 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 by shifting the member *ctlm_offsethi* to the left by thirty two and then doing a binary or of *ctlm_offsetlo*.

Encoding of Enumerations

Enumerations, noted by the type **CTF_K_ENUM**, are similar to structures. Enumerations use the variable list to note the number of values that the enumeration contains, which we'll term enumerators. In C, an enumeration is always equivalent to the intrinsic type **int**, thus the value of the member *ctt_size* is always the size of an integer which is determined based on the current model. For FreeBSD systems, this will always be 4, as an integer is always defined to be 4 bytes large in both **ILP32** and **LP64**, regardless of the architecture. For further details, see arch(7).

The enumerators encoded in an enumeration have the following structure in the variable list:

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

The member *cte_name* refers to the name of the enumerator's value, it is encoded according to the rules in the section *String Identifiers*. The member *cte_value* contains the integer value of this enumerator.

Encoding of Forward References

Forward references, types of kind **CTF_K_FORWARD**, in a **ctf** file 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 member of the **struct ctf_type_v3** that matters for a forward declaration is the *ctt_name* which points to the name of the forward reference in the string table as described earlier. There is no other information recorded for forward references.

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

Pointers, typedefs, volatile, const, and restrict are all similar in **ctf**. They 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 use the member *ctt_type* to refer to the base type that they modify.

Encoding of Unknown Types

Types with the kind **CTF_K_UNKNOWN** are used to indicate gaps in the type identifier space. These entries consume an identifier, but do not define anything. Nothing should refer to these gap identifiers.

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 last section of the **ctf** file is the **string** section. This section encodes all of the strings that appear throughout the other sections. It is laid out as a series of characters followed by a null terminator. Generally, all names are written out in ASCII, as most C compilers do not allow any characters to appear in identifiers outside of a subset of ASCII. However, any extended characters sets should be written out as a series of UTF-8 bytes.

The first entry in the section, at offset zero, is a single null terminator to reference the empty string. Following that, each C string should be written out, including the null terminator. Offsets that refer to something in this section should refer to the first byte which begins a string. Beyond the first byte in the section being the null terminator, the order of strings is unimportant.

Data Encoding and ELF Considerations

ctf data is generally included in ELF objects which specify information to identify the architecture and endianness of the file. A **ctf** container inside such an object must match the endianness of the ELF object. Aside from the question of the endian encoding of data, there should be no other differences between architectures. While many of the types in this document refer to non-fixed size C integral types, they are equivalent in the models **ILP32** and **LP64**. If any other model is being used with **ctf** data that has different sizes, then it must not use the model’s sizes for those integral types and instead use the fixed size equivalents based on an **ILP32** environment.

When placing a **ctf** container inside of an ELF object, there are certain conventions that are expected for the purposes of tooling being able to find the **ctf** data. In particular, a given ELF object should only contain a single **ctf** section. Multiple containers should be merged together into a single one.

The **ctf** file should be included in its own ELF section. The section's name must be '.SUNW_ctf'. The type of the section must be **SHT_PROGBITS**. The section should have a link set to the symbol table and its address alignment must be 4.

SEE ALSO

ctfconvert(1), ctfdump(1), ctfmerge(1), dtrace(1), elf(3), gelf(3), a.out(5), elf(5), arch(7)