NAME

ctf – Compact C Type Format

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.

This man page discusses high level **CTF** concepts, and describes the process of adding **CTF** to objects. Other man pages document the format, and the utilities that manipulate it.

ctf(5)	on-disk **CTF** format
ctfconvert(1)	add **CTF** to relocatable objects
ctfdump(1)	dump CTF content
ctfmerge(1)	merge **CTF** from multiple objects
ld(1)	**-z ctf**: perform **CTF** convert/merge as part of link-edit

Compilers produce debug sections for the objects they create. These debug sections are valuable for debuggers, and particularly for source level debuggers, but they can be very large, and are often omitted from production software used in non-debug environments. In contrast, **CTF** data is generally small enough to be kept in almost any object, while providing basic type information valuable to low level observability tools. The mdb(1) and kmdb(1) debuggers, dtrace(8), and the proc(1) tools, all make use of **CTF**, when present, to enhance their operation.

There are 2 stages in the process of adding **CTF** to objects: generation, and merging.

Generation

CTF is added when source code is compiled into relocatable objects. This process is often called **conversion**, as **CTF** is often produced by reading the type information from the debug sections produced by the compiler to extract the type information, thereby converting it to **CTF**. **CTF** is typically generated by requesting debug data from the compiler, and applying the **ctfconvert** utility to the resulting object. Having served its purpose, the debug data may then optionally be stripped as part of this operation.

```
% cc -c -g main.c
% ctfconvert -S main.o
```

The **ctfconvert** utility is the only option for generating **CTF** for objects compiled with the native Studio compilers, and is compatible with the debug data produced by that compiler's **-g** option. There is also support for converting the **DWARF** sections produced by the **gcc** compilers. However, this support is limited to version 2 **GNU DWARF**, which is not the current version produced by those compilers.

```
% gcc -c -g -gdwarf=2 main.c
```

```
% ctfconvert -S main.o
```

A better option for users of **gcc** is to bypass the use of **ctfconvert** entirely, and specify the **-gctf** option to generate **CTF** directly.

```
% gcc -c -gctf main.c
```

The **CTF** produced by **gcc** is a different variant of **CTF** that is not directly compatible with the native **CTF** used on Oracle Solaris. However, the Solaris **CTF** implementation is able to recognize **GNU CTF**, and will automatically translate it to the native form when it is read during the merge stage. See GNU CTF.

Merging

When objects are linked together to form a final object, such as an executable, shared object, or kernel module, the **CTF** from the input objects must be merged to form a final single **CTF** section that describes the resulting object. The **ctfmerge** utility is used to carry out the merge step. A small complete example, including the compile and link, might look as follows.

```
% cc -c -g main.c
% ctfconvert -S main.o
% cc -o main main.o
% ctfmerge -o main main.o
```

It is not sufficient to simply link the program, and omit the merge step. While the resulting object will contain a **.SUNW_ctf** section formed from the concatenation of the input **CTF** data, that section is not valid **CTF**, and is not usable. The **ctfmerge** utility reads the **CTF** data from the input objects, merges them to form a description of the complete output object, and rewrites the object, replacing the contents of the **.SUNW_ctf** section with valid CTF.

The required sequence of commands can be simplified, by omitting the use of ctfconvert for each input object, and instead, specifying the **-a** option to ctfmerge to have it perform the conversion step for each object before doing the merge.

```
% cc -c -g main.c
% cc -o main main.o
% ctfmerge -a -o main main.o
```

An even larger simplification results from using the **ld -z ctf** option to incorporate the **CTF** convert and merge steps into the link-edit. This is very close to the commands required to build a program without CTF, and is the simplest way to incorporate **CTF** into most software.

```
% cc -c -g main.c
% cc -o main main.o -zctf=convert
```

The Solaris **CTF** implementation is able to recognize **GNU CTF**, and will automatically translate it to the native form when it is read during the merge stage. See GNU CTF. This support allows the **gcc** version of the example to be a single command.

```
% gcc -gctf main.c -zctf
```

The **-gsctf** option to the **gcc** compiler can be used to reduce this to a single option. When **-gsctf** is specified, **gcc** passes the **-gctf** option to the compiler, and **-z ctf** to the link-editor.

```
% gcc -gsctf main.c
```

GNU CTF

The **GNU gcc** compilers, and supporting **binutils** toolchain components, including the **GNU** link-editor, support a different version of **CTF**, the design of which evolved from the Solaris version. Despite that connection, the two are distinct and incompatible formats. The **GNU CTF** produced by the **-gctf** option to **gcc** cannot be used directly on Solaris. However, the two formats are similar enough that one can be translated to the other. The merge support provided by the Solaris **CTF** implementation is able to recognize **GNU CTF**, and will automatically translate it to Solaris compatible **CTF** on input.

The **ctfdump** utility is able to dump **GNU CTF**, and can be used to inspect its details. The result of the conversion to Solaris **CTF** can be seen by applying **ctfdump** to the final object that includes this object in the link, or more directly, by applying **ctfconvert -G** to the object compiled with **gcc -gctf**. Note that it is not necessary to use **ctfconvert -G** on objects compiled with **gcc**, since the merge process will automatically translate them. The **-G** option exists primarily to facilitate inspection with **ctfdump**.

Solaris **CTF** is written to **ELF** objects in a section named **.SUNW_ctf**, of type **SHT_SUNW_CTF**. In contrast, the **GNU CTF** produced by **gcc** is written to a section named **.ctf**, of type **SHT_PROGBITS**. As such, the 2 formats are able to coexist. When a **.ctf** section is encountered by the Solaris **CTF** merge implementation, the data is automatically translated to a form compatible with Solaris, merged with the data from other objects, and the result is written as a native Solaris **.SUNW_ctf** section, fully compatible with tools such as **mdb** and **dtrace**.

The **gcc -gctf** option is known to be well supported for the C language. Support for some C++ language levels may not be present. See CTF Coverage and Language Compatibility.

CTF Coverage And Language Compatibility

The **CTF** format describes types at the level of the type system of the C programming language. As such, it is most useful when applied to programs written in C, or languages with type systems similar to C. This is sometimes called a **machine level view**.

Languages with more complex type systems can be used with **CTF**, but the **CTF** will correspond to the basic building blocks from which those more complex types are constructed, and the correspondence between the two may not be obvious. A significant example is that of C++. **CTF** can be used with C++, and will properly represent the machine level types from which abstractions such as classes are formed. However, those higher level concepts will not be visible in the **CTF**, which can limit its value.

Ideally, every input object used to build an executable, shared object, or kernel module, contains **CTF** describing its contents. When linked together into a resulting final object and merged, the resulting **CTF** will provide full coverage for all types used within that object. The **ctfmerge -t** option, or **ld -zctf=require**, can be used to enforce this, and guarantee full coverage. Otherwise, the coverage will depend on the objects being linked. Similarly, and related to the discussion of languages other than C above, the **ctfmerge -i** option, or **ld -zctf=ignore-non-c**, can be used to exclude non-C code from this requirement. Not requiring **CTF** in all input objects may allow the **CTF** in a program that has still has high coverage to succeed, which is a useful outcome. Conversely, it can allow a program with low coverage to produce **CTF** that is too incomplete to be useful. The programmer must use their knowledge of the code being built to determine whether useful **CTF** can be produced for a given code base, and to decide whether and how to use these options to best effect.

Compression

CTF data may be compressed to reduce the space used in the object. For historical reasons, there are 2 distinct forms of compression. The first employs the **ZLIB** based compression provided by the **CTF** format. See **ctf**(5). The second employs the general section compression mechanism provided by the ELF object format. See **elf_compress**(3elf), and the *Oracle Solaris 11.4 Linkers and Libraries Guide*. The **CTF** specific form predates the introduction of general section compression features to **ELF** by over a decade. The **CTF** variant is based on **ZLIB**, while the **ELF** form offers a choice of **ZLIB** or **ZSTD**. Both approaches are fully supported, and deliver similar performance. In the future, new compression options may be delivered through the **ELF** mechanism, while the compression features of the original **CTF** version are not expected to change.

Compression can be applied to **CTF** data in a variety of ways.

ctfconvert / ctfmerge

By default, the **ctfconvert** utility does not compress the **CTF** it produces, as the size of the **CTF** in a single compilation unit is typically very small. In contrast, the **ctfmerge** defaults to compression using the original **CTF** form of **ZLIB** based compression. Both commands provide the **-z** option to select other compression options. The syntax of the option is, **-z** *cmp-type*[,*cmp-opt*...], where *cmp-type* is one of the following.

none

The **CTF** data is not compressed.

ctf

The **CTF** data is compressed using the original **CTF** form of **ZLIB** based compression. In this case, the resulting **CTF** data will have the **CTF_F_COMPRESS** flag set in the **CTF** header.

zlib / zstd

The **CTF** data is compressed using the **ELF** form of **ZLIB**, or **ZSTD**, based compression. When the **ELF** mechanism is used, the section header for the **.SUNW_ctf** section will have the **SHF_COMPRESSED** flag set.

The **ELF** form of compression provided by **zlib** and **zstd** only compress the data if the result will be resulting section size will be smaller than the uncompressed data. The **force** *cmp-opt* can be used with those options to force compression to be done unconditionally.

Link-Editor (ld)

The **ld -z ctf** option can be used to add **CTF** to objects at link time. The resulting **CTF** data is not compressed by default. This data can be compressed using the original **CTF** form of ZLIB based compression by specifying the **-z ctf=compress** option. Alternatively, the **-z compress-class=ctf** option can be used to employ the **ELF** compression mechanism. It is recommended that only one of these options be used, and not both together. Double compression is slow, and will typically produce larger results.

elfcompress

The **elfcompress** utility can be used to compress or decompress **CTF** data in existing objects that employ the **ELF** form of compression.

Ancillary Objects

Ancillary objects are a link-editor feature that allow the non-allocable sections associated with an object to be written to one or more separate objects. Non-allocable sections are not required at runtime, and are primarily used by debuggers and other observability tools. Ancillary objects are created by the link-editor when the **ld -z ancillary** option is specified, or when specified by a mapfile. See **ld**(1), and the *Oracle Solaris 11.4 Linkers and Libraries Guide*.

Ancillary objects are typically only created by the link-editor for shared objects and executables. They are not created by compilers when producing relocatable objects, which are intended to be later linked into final objects. For this and other reasons, ancillary objects are not supported as input for **CTF** generation, and are therefore rejected by **ctfconvert**, as input objects to **ld -z ctf**, and as input to **ctfmerge**.

Ancillary objects are supported as the destination for merged **CTF** data.

ctfmerge

All ancillary objects must be present in the same directory as the specified primary object. If the objects have an existing **.SUNW_ctf** section, those sections that do not have the **SHF_SUNW_ABSENT** section flag set are rewritten with the newly merged **CTF** data. If the objects do not have an existing **.SUNW_ctf** section, one is added. In this case, the section in the primary object receives the merged data, and the sections in the ancillary objects are all marked as **SHF_SUNW_ABSENT**.

ld -z ctf

.SUNW_ctf sections containing the merged **CTF** data are added to the output objects in the usual manner, and a mapfile can be used to control the placement within ancillary objects.

Objects that have associated ancillary objects can be inspected with the **ctfdump** utility. If the object has associated ancillary objects, and those ancillary objects are available in the same directory as the primary object, **ctfdump** will transparently read those ancillary objects to obtain any sections absent from the primary object which are needed to display the CTF. Support for reading ancillary objects is limited to plain objects. The **ctfdump** utility will not access ancillary objects for objects found in an archive.

Labels

Labels can be associated with **CTF** data. Labels can be useful for identification purposes, but are optional, unless **uniquification** is desired, in which case labels must be provided. When using the **ctfmerge** and **ctfconvert** utilities, labels are specified using the **-l**, or **-L** options. When using **ld -z ctf**, labels are specified with the **label**, or **label-env** suboptions.

Uniquification

When multiple objects share common type definitions provided by a central core object, the size of the overall **CTF** data can be greatly reduced through the process of **uniquification**. Uniquification removes definitions found the core object from other objects, leaving those other objects with only the additional definitions that are unique to them. The core object is usually referred to as the **parent**, and the other object as the **child**. A given child can only have one parent, and the parent/child relationship is only one level deep, with no further descendants. The **CTF** data in the parent and child objects to be uniquified should define a common label, identifying them as sharing common type definitions.

When using the **ctfmerge** utility, uniquification is specified using the **-d** and **-D** options. When using **ld -z ctf**, uniquification is specified with the **uniqify-file**, and **uniqify-label** suboptions.

Uniquification is typically applied only to kernel modules. In the kernel environment, the **genunix** kernel module is the parent, and the other kernel modules delivered with the system are uniquified against it. For non-kernel objects, the benefits of unification are minor, the overhead in terms of management complexity significant, and unification is not recommended.

Additive Merges

There are cases where it is desired to issue a new version of an object that has an existing uniquification relationship to another object. A common example occurs when operating system kernel modules are patched. For this to work smoothly, it is necessary to preserve all preexisting **CTF** data, unchanged, while adding any necessary additional definitions needed by the replacement object. This operation is known as an **additive merge**. In the case of an additive merge, a final uniquification is performed against the **CTF** data in the previous version of the module. The result is the placement of new and changed data after the existing data, thus preserving the existing type definitions.

When using the **ctfmerge** utility, an additive merge is done using the **-w** option. When using **ld -z ctf**, an additive merge is specified with the **additive-merge** suboption.

EXAMPLES

The following examples demonstrate the options for adding **CTF** data to a program named **prog**, built from 3 source files, **main.c**, **sub1.c**, and **sub2.c**. In each case, the options for the native **cc** compiler, and **gcc**, are shown.

Example 1 Add CTF Using **ctfconvert** and **ctfmerge**

The most basic way to add **CTF** to a program is to use **ctfconvert** to add **CTF** to each source file as it is compiled into a relocatable object, and then to apply **ctfmerge** to the resulting program.

```
# cc
% cc -c -g main.c
% ctfconvert -S main.o
% cc -c -g sub1.c
% ctfconvert -S sub1.o
% cc -c -g sub2.c
% ctfconvert -S sub2.o
% cc -o prog main.o sub1.o sub2.o
% ctfmerge -o prog main.o sub1.o sub2.o
```

When using **gcc**, the need for the **-g** option goes away, and the **-gctf** option is used instead of running **ctfconvert** .

```
# gcc
% gcc -c -gctf main.c
% gcc -c -gctf sub1.c
% gcc -c -gctf sub2.c
% gcc -o prog main.o sub1.o sub2.o
% ctfmerge -o prog main.o sub1.o sub2.o
```

Without the need to run **ctfconvert** on each relocatable object, the **gcc** version can be further reduced to a single invocation of the compiler to compile all source files in a single call. Note that the compile and link steps must still be kept separate, as **ctfmerge** needs to examine each input object individually. .

```
# gcc
% gcc -c -gctf main.c sub1.c sub2.c
% gcc -o prog main.o sub1.o sub2.o
% ctfmerge -o prog main.o sub1.o sub2.o
```

Example 2 Add CTF Using **ctfmerge -a**

The **-a** option to **ctfmerge** can be used to simplify the previous example, by removing the requirement to run **ctfconvert** on each input object. A convert operation is still needed, but in this version, is done by **ctfmerge** as each object enters the merge operation. When a program is built once, the cost of these two approaches is identical. In a case where the code is being modified and built repeatedly as part of code development, possibly driven by the **make** utility, it can be marginally more expensive, as the convert step is done for every input object after every link, as opposed to once when each input object is recompiled. This cost may be noticed when working on very large code bases, but probably not otherwise. Conversely, the required **Makefile** rules will be simpler.

```
# cc
% cc -c -g main.c
% cc -c -g sub1.c
% cc -c -g sub2.c
% cc -o prog main.o sub1.o sub2.o
% ctfmerge -a -o prog main.o sub1.o sub2.o
```

While the **-a** option to **ctfmerge** can be used with **gcc** as well, it provides no benefit when the **-gctf** option to **gcc** is used, as the resulting input objects are created with **CTF**, and do not need **ctfmerge** to generate it.

For smaller programs, this can be simplified further, by passing all source files to a single invocation of the compiler. Note that compile and link steps must still be kept separate, as **ctfmerge** needs to examine each input object individually.

```
# cc
% cc -c -g main.c sub1.c sub2.c
% cc -o prog main.o sub1.o sub2.o
% ctfmerge -a -o prog main.o sub1.o sub2.o
```

Example 3 Add CTF Using **ld -z ctf**

The link-editor can be used to simplify the addition of **CTF** by dropping the use of **ctfconvert** and **ctfmerge**, and instead using the link-editor's **-z ctf** option to invoke those operations from within the link-edit. The cost of this approach is similar to that of the previous example which used **ctfmerge -a**. Note that the **cc** version requires the use of the **convert** suboption to **-z ctf**, while the **gcc** version does not.

```
# cc
% cc -c -g main.c
% cc -c -g sub1.c
% cc -c -g sub2.c
% cc -o prog main.o sub1.o sub2.o -zctf=convert

# gcc
% gcc -c -gctf main.c
% gcc -c -gctf sub1.c
% gcc -c -gctf sub2.c
% gcc -o prog main.o sub1.o sub2.o -zctf
```

For smaller programs, this can be simplified further, by passing all source files to a single invocation of the compiler.

```
# cc
% cc -c -g main.c sub1.c sub2.c
% cc -o prog main.o sub1.o sub2.o -zctf=convert

# gcc
% gcc -c -gctf main.c sub1.c sub2.c
% gcc -o prog main.o sub1.o sub2.o -zctf
```

Unlike the version that employs the **-a** to **ctfmerge**, the compile and link do not need to be kept separate, so this can be reduced to a single operation.

```
# cc
% cc -g -o prog main.c sub1.c sub2.c -zctf=convert

# gcc
% gcc -gctf -o prog main.c sub1.c sub2.c -zctf
```

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.

Support for reading **GNU CTF** from relocatable objects created by the **gcc** compilers and translating it to the native **CTF** representation was added in the Oracle Solaris 11.4.84 release.

SEE ALSO

ctfconvert(1), **ctfdump**(1), **ctfmerge**(1), **ld**(1), **mdb**(1), **libz**(3), **ctf**(5), **dtrace**(8)

Oracle Solaris 11.4 Linkers and Libraries Guide