Building applications using ANDF

Stavros Macrakis

An ANDF-based application is built in two main steps: building target-independent modules, and building target-dependent load files from them and target-dependent modules. This paper shows how.

1. Introduction

The standard C compiler `cc` translates a set of C source modules into a runnable program.

The ANDF C compiler `tcc` consists of two parts. The C producer `tdfc` translates portable C source modules[1] into ANDF[2]; the ANDF installers *x*`trans` translate ANDF into a runnable program.

In both cases, the compilation depends on system libraries. But unlike traditional C, ANDF C permits specifying fully abstract interfaces separately from their machine-dependent instantiations.

A distributable ANDF module is built by an ANDF producer with an abstract environment such as POSIX. A runnable application is built by an installer from the distributable module and the concrete system library interfaces and implementations which instantiate that abstract environment.

1. See the paper *Porting to ANDF* for a discussion of portable programming issues.
2. See the paper *The Structure of ANDF* for a discussion of the internal form of ANDF.

This paper shows how to use ANDF tools to build applications. We concentrate on the common, important cases: many other facilities are documented in the `man` pages.

2. *Basic use of* `tcc`

`tcc` is the driver for the ANDF tools. It can be thought of as an ordinary compiler which happens to use ANDF as its intermediate language. In fact, `tcc` can be used as a plug-compatible replacement for the native `cc` compiler:

```
> tcc -Ysystem prog.c
> a.out
```

-Ysystem specifies that the system's own header files will be used to define the programming environment.

More interestingly, `tcc` can stop after producing the ANDF form:

```
> tcc -Ysystem -Fj prog.c
```

`-Fj` stops the compilation process after producing the `.j` file

This produces the machine-independent file `prog.j`, which can then be translated to a runnable program:

```
> tcc prog.j
```

(Like `cc`, `tcc` uses the file type as encoded in the filename extension to determine the appropriate operation.[3])

Compiling with the `-Ysystem` flag abstracts from the host machine's instruction set, but not the programming environment defined by its header files, so in general the `.j`'s will not be installable on other platforms.

3. See "File types" (p. 3) for more details.

3. Environments

Ordinary C compilers implicitly use the environment of the host machine as the environment for the program. This is true not only for data representations and instruction set, but also for system libraries.[4]

The `tcc` compiler allows explicit choice of the environment using the `-Y` flag. Currently supported standard environments include:

`ansi`	Strict ANSI C, including the ANSI C library. This is the default. Note that the ANSI C library provides only minimal file primitives, and no directory primitives, in order to be independent of the operating system.
`posix`	The environment defined by POSIX P1003.1, a superset of the `ansi` environment.
`xpg3`	The environment defined by the X/Open Portability Guide, version 3, a superset of the `posix` environment.

Besides environments defined by standards, `tcc` also supports the native environment:

`system`	The local (native) environment.

Although the system environment may be a superset of a standard environment, it also allows the application to depend on implementation choices, extensions, or even errors in the interface. Thus, it should be avoided when developing portable programs.

A portable application compiled in a given *standard* environment should install correctly on all platforms supporting that environment.[5]

An ANDF environment defines available library interfaces and the minimal acceptable run-time environment (*e.g.* integer range). Standard environments are defined abstractly, which means that different concrete realizations are possible.

4. Most compilers provide some mechanism for overriding the default include and load paths which define the environment.

5. See our paper on environment validation (forthcoming).

4. Concrete and abstract interfaces

ANDF defines interfaces abstractly, and specifies their concrete form and implementation separately. C supports separating the concrete interface and the implementation for most entities (prototype declarations and external variables), but not all (notably macros). And C provides no mechanism for separating the abstract interface and the concrete interface.

For instance, many file operations in ANSI C require an argument of type `FILE`. C header files define `FILE` as, *e.g.*:

```
typedef
     struct {
        unsigned char *_ptr;   /* accessible to all users of FILE */
        int _cnt;              /* accessible to all users of FILE */
        ...
           }
     FILE;
```

That `FILE` is a type (`typedef` … `FILE`;) is an important part of the abstract interface; the fact that it is a structure with certain fields (the italicized text) is part of the concrete interface, and should not be visible to the application, since it may well vary from implementation to implementation. Although convention dictates that fields beginning with "_" are private, the language and environment provide no support for this distinction.

Newer programming languages support this distinction in the language and environment. For instance, Ada divides package specifications into a public part (abstract interface), and a private part (concrete interface):

```
package stdio is
   type FILE is private; ...
private
   type FILE is
      record
         ptr : int;        — Not visible outside package
         cnt : int;        — Not visible outside package
         ...
      end record;
end stdio;
```

ANDF C goes beyond this, and provides a pragma extension to C which allows specifying the abstract and the concrete interfaces in completely different places. It defines `FILE` as an abstract type in the producer's header files (similar to the public part of a specification), and defers its implementation to the installer's header files (similar to the private part of a package specification).[6]

5. Building and dependencies

An application must refer to the abstract interface when it is translated by the producer, the concrete interface when it is compiled by the installer, and the implementation when it is linked.

In C, modules are referred to by filenames in `include` statements. `tcc` uses the environment/filename pair to identify a particular abstract interface. At installation, the package must link to particular concrete interfaces. This is done by `tcc` using `tld`. Finally, after translation to .o form, tcc links the module to native libraries using `ld`.

6. Using `tcc` *for application delivery*

For delivering applications to users, it is desirable to package the target-independent .j files into an archive. This is done with the `-prod` flag, which creates a single target-independent ANDF archive. Besides linking with platform-dependent token definition libraries, the installation script may wish to use target-dependent object code (`.o`), typically derived from assembly code.

In order to discourage reverse engineering, it is usually desirable to suppress all identifier names. This is done with the -MA flag.

6. For more details, see the paper *Porting to ANDF*.

7. Using `tcc` for libraries

Unlike applications writers, library writers need to provide external interfaces which can be linked to at installation time, just like standard environments. These should be abstract interfaces, and can be defined using ANDF C's abstraction mechanisms.

The `-hlib` flag of the ANDF linker `tld` allows delivering platform-independent code with only the required interfaces exposed at install time.

8. Dialects of C

`tcc` supports various dialects of C. Although we recommend using ANSI C (the default), older code can be compiled using the `-X`*x* flags. Finer control is possible with the `-not_ansi` and `-nepc` (no extra portability checks) flags. Even finer control of the implementation-defined features of ANSI C is available using specialized pragmas, but this should not be necessary.

Since using ANSI C and retaining full portability checking gives a higher level of confidence in application portability, we recommend using ANSI C.

9. File types

`tcc`, like standard C compilers, distinguishes various forms of a program by their filename extension. In addition to `.c` (C source), `.h` (header files), `.i` (preprocessed C source), `.o` (object), `.a` (object archive), and `.out` (runnable load file), tcc also defines:

- `.j` target-independent ANDF
- `.ta` archives of target-independent ANDF
- `.t` target-dependent ANDF
- `.tl` target-dependent (concrete) interfaces defined in ANDF

Appendix A. `tcc` *man page*

Crown Copyright, 1993

NAME

`tcc` user interface to the TDF system

SYNOPSIS

`tcc` [options] files

DESCRIPTION

`tcc` is the user interface to the TDF system. It accepts several types of arguments as files:

- Arguments whose names end in `.c` are understood to be C source files.
- Arguments whose names end in `.i` are understood to be preprocessed C source files.
- Arguments whose names end in `.j` are understood to be target independent TDF capsules.
- Arguments whose names end in `.ta` are understood to be archives of target independent TDF capsules.
- Arguments whose names end in `.t` are understood to be target dependent TDF capsules.
- Arguments whose names end in `.s` are understood to be assembly source files.

All other arguments (but particularly those whose names end in `.o` and `.a`) are understood to be binary object files.

The compilation process is as follows. Depending on the options given, it may halt at any stage:

TDF archives are split into their constituent target independent capsules.

C source files (including preprocessed C source files) are compiled into target independent TDF capsules using `tdfc`(1tdf).

Target independent TDF capsules are linked, using `tld`(1tdf), with the TDF libraries to produce target dependent TDF capsules.

Target dependent TDF capsules are translated into assembly source files using one of `mipstrans`(1tdf) (*q.v.*—things are not quite so simple in this case), `vaxtrans`(1tdf) *etc.*

Assembly source files are compiled into binary object files using `as`(1).

Binary object files are linked with the precompiled libraries, using `ld`(1), to produce a final executable.

With the exception of binary object files, intermediate files are not preserved unless explicitly instructed.

The standard file suffixes, `c`, `i`, `j`, `t`, `s`, and `o`, together with `p` (pretty-printed TDF capsule), are used to indicate file types in some options. Also the various compilation phases are identified by letters in some options. These are :

c	C to TDF producer
p	Preprocessor
L	TDF builder (or linker)
t	TDF translator
a	System assembler
l	System linker
d	TDF pretty-printer

OPTIONS

The following options are supported by `tcc`. All options are scanned before input files are dealt with.

`-Bstatic`	Tells the system linker to link statically.
`-Bdynamic`	Tells the system linker to link dynamically.
`-D`*string*	Where string is of the form *macro* or *macro=defn*, is equivalent to inserting the preprocessor directives `#define` *macro* `1` or `#define` *macro* `defn` at the start of each C source file. This is implemented by `tcc` writing this directive into a start-up file.
`-E`	Invokes the C preprocessor only, putting the result into a file with a `.i` suffix if other options indicate that preprocessed C files are to be preserved, or onto the standard output otherwise.
`-F`*letter*	Tells `tcc` to stop after producing the files indicated by *letter*, and to preserve these files. *letter* is a single character corresponding to the suffix of the files to be preserved.
`-I`*string*	Tells the C to TDF producer to search the directory *string* for included files. The producer searches the directories in the order given, followed by the system default directories.

`-J`*string* — Tells the TDF builder to search the directory string for TDF libraries. The builder searches the directories in the order given, followed by the system default directories.

`-L`*string* — Tells the system linker to search the directory *string* for libraries. It searches the directories in the order given, followed by the system default directories.

`-M` — Specifies that the TDF builder should link all the given target independent TDF capsules into one. This is done between stages 2 and 3 above. The default name for the produced capsule is `a.j`.

`-MA` — Specifies that the TDF builder should link all the given target independent TDF capsules into one and also hides all the defined tag and token names from the resultant TDF (except “main”). This should only be used to compile complete programs. The default name for the resulting capsule is `a.j`.

`-Map_letter_to_letter`
Not documented.

`-O` — This flag has no effect other than to cancel any previous diagnostics flag and is included only for compatibility with other compilers. All TDF optimizations are on by default. All optimizations are believed to be correct, any bug which occurs in the fully-optimized state is a genuine bug.

`-P` — Invokes the C preprocessor only, putting the result into a file with a `.i` suffix if other options indicate that preprocessed C files are to be preserved, or onto the standard output otherwise.

`-Pa` — Preserves all intermediate files.

`-P`*letter...* — Tells `tcc` to preserve those files indicated by *letter*. Each letter is a single character corresponding to the suffix of the files to be preserved. The `tcc` startup-file can be preserved as `tcc_startup.h` using `h`.

`-S` — Tells `tcc` to stop after producing an assembly source file.

`-S`*letter*`,`*string,...* — The specifies that the list of input files *string* all have type *letter*, where *letter* is a single character giving the normal suffix of the file type. This gives an alternative method of passing input files to `tcc`, one which does not depend on it having to recognise suffixes to find the type of a file.

`-U`*string* — Is equivalent to inserting the preprocessor directive `#undef` *string* at the start of each C source file. This is implemented by `tcc` writing this directive into a start-up file. The only macros built into the C to TDF producer are `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__` and `__STDC__`.

`-W`*letter*`,`*string,...* — This passes the list of options *string* to the compilation phase indicated by *letter*.

`-Xa` — This option specifies that the given program is written in ANSI C. This is default.

`-Xc` — This option specifies that the given program is written in "K&R" C.

`-Xt` — This option specifies that the given program is written in "traditional" C.

`-Y`*string* — Specifies the environment to use. An environment is a file telling `tcc` to modify its defaults. If the full pathname of env is not given, the file is searched for along the `tcc` environments path which is a list of directories separated by colons. There are certain standard environments, for example, `ansi`, representing the ANSI standard, is the default environment, `posix` represents the POSIX standard, and `xpg3` the XPG3 standard. The `system` environment allows `tcc` to behave like `cc`(1), using the system header files etc. See `tcc_env`(1tdf) for more details about environments.

`-b` — Stops the library `libc.a` being used by the linker by default.

`-c` — Tells `tcc` to stop after producing the binary object files.

`-clean` — By default, if more than once binary object file is produced during the compilation process, they are all preserved. This option overrides this.

`-dry` — Makes `tcc` print information on what system commands it would execute with the given files and options (as in verbose mode) but not actually perform them.

`-disp` — Runs the TDF pretty-printer on all files at stage 2 or 3 and then terminates. The results are put into files with `.p` suffixes.

`-disp_t` — Runs the pretty-printer on all files at stage 3 and then terminates. This differs from the previous option in that it displays the TDF after linking with the target-dependent TDF libraries rather than before. The output in put into a file with a `.p` suffix.

`-e`*string* — Is equivalent to inserting the preprocessor directive `#include` *string* at the end of each C source file.

`-f`*string* — Is equivalent to inserting the preprocessor directive `#include` *string* at the start of each C source file.

`-g` — Tells `tcc` to produce diagnostic information compatible with the system debugger.

`-hide_names` — Tells `tcc` to hide capsule names when constructing a TDF archive.

`-i` — Tells `tcc` to stop after producing the target independent TDF capsules.

`-j`*string* — Tells the TDF builder to use the TDF library `string.tl`.

`-keep_errors` — By default, if an error occurs during the production of a file, `tcc` will remove it. This option will preserve such files.

`-l`*string* — Tells the system linker to use the library `libstring.a`.

`-make_up_names`
Causes `tcc` to make up names for all intermediate files rather than forming them from the basenames of the input files.

`-no_startup_options`
Not documented.

`-nepc` — Tells the C to TDF producer to allow certain non-portable constructs through.

`-not_ansi` — Tells the C to TDF producer to allow certain non-ANSI features through.

`-o`*string* — If a final executable is produced, call it *string* (the default is `a.out`). Otherwise, if only one file is preserved, call it *string*.

`-p` — Produces profiling data for use with `prof`(1) on those machines for which this command is available.

`-prod` — Specifies that `tcc` should stop after producing the target-independent TDF capsules and combine them into a TDF archive. The default archive name is `a.ta`.

`-q`
`-quiet` — Specifies that `tcc` should work silently. This is default.

`-show_errors` — Makes `tcc` report on the command it was executing when an error occurred.

`-show_env_path`
: Prints the `tcc` environments path. This is a list of directories separated by colons. The first element of the list is always the system default environments directory and the last element is always the current working directory. The other elements may be specified by the user by means of the `TCC_ENV` system variable.

`-target`*string*
: No effect (allowed for compatibility with `cc`(1) on some systems).

`-time`
: Makes `tcc` print information on what system commands it is executing (as with verbose mode) followed by the time taken for each.

`-verbose`
: Specifies that `tcc` should work in verbose mode, sending information on what system commands it is executing to the standard output.

`-version`
: Makes `tcc` report its version number.

`-w`
: Suppresses all `tcc` warning messages.

`-work`*string*
: Specifies that all preserved intermediate files are placed in the directory *string* rather than where they are placed by default, in the current working directory.

`-wsl`
: Tells the C to TDF producer to make all string literals writable.

FILES

file`.c`	C source file
file`.i`	Preprocessed C source file
file`.j`	Target independent TDF capsule
file`.t`	Target dependent TDF capsule
file`.s`	Assembly source file
file`.o`	Binary object file
file`.p`	Pretty-printed TDF capsule
`a.out`	Default executable name
`a.ta`	Default TDF archive name
`a.j`	Default output file for merge-TDF-capsules option
`tcc_startup.h`	Name of preserved tcc start-up file

`/tmp/tcc*` Temporary directory (this may be changed using the TMPDIR system variable, see tempnam(3)).

SEE ALSO

`as`(1), `cc`(1), `disp`(1tdf), `ld`(1), `prof`(1), `tcc_env`(1tdf), `tdf`(1tdf), `tdfc`(1tdf), `tld`(1tdf), `trans`(1tdf).

Appendix B. tld man page

Crown Copyright, 1993

NAME

tld TDF linking and library manipulation utility

SYNTAX

`tld [-ml] [` *switches*`]...`

`tld -mc [` *switches* `]`

`tld -mt [-verbose]` *library*

`tld -mx [-nc] [-verbose]` *library capsule-name*`...`

`tld -mv`

DESCRIPTION

The `tld` command is used to create and manipulate TDF libraries, and to link together TDF capsules. It has four modes, selected by one of the `-ml` (link TDF capsules), `-mc` (create TDF library), `-mt` (list library contents) or `-mx` (extract capsules from library) switches. The `-mv` switch can be used to find out the version number of `tld` being used. If provided, these switches must be the first on the command line. If one is not provided, the `-ml` switch is assumed. The different modes are described below. In the description, tag definitions are referred to as either unique or multiple. A unique definition is a definition where the defined attribute is set; a multiple definition is one where the multiple attribute is set (*i.e.* more than one definition is allowed). A definition may be both multiple and unique (if both bits are set).

LINKING

In the default mode, `tld` tries to link together the TDF capsules specified on the command line. This consists of the following stages:

1. All of the capsules specified on the command line are loaded, and their token, tag and other identifiers are mapped into a per-identifier-type namespace. In these common namespaces, all external tokens with the same name will be mapped to the same identifier. The same is true for external tags. During this process, `tld` will report errors about any attempt to link together more than one capsule providing a definition for any token or tag (in the case of tags, the capsules will link successfully as long as no more than one of them is a unique definition).

2. If any libraries were specified on the command line and there are tokens or tags which are used but not defined in the capsules, then the libraries are scanned to see if they can provide the required definitions. Any cap- sules that provide necessary definitions are loaded. There must only be one definition for each token or tag in all of the libraries (in the case of a tag, this may be either one non-unique definition, or one unique definition with zero or more non-unique definitions; if a unique definition exists, then the non-unique definitions are ignored).

3. If any tokens or tags require hiding (specified by command line switches), then they are hidden at this point. Hiding means removing the name of the token or tag from the external name list. It is illegal to hide undefined tokens or tags.

4. A new TDF capsule is created, consisting of all of the input capsules and the necessary library capsules. Unless specified with the `-o` switch, the output file will be called `tdf-output.t`.

Switches

Tld accepts the following switches in link mode.

`-debug` Produce a diagnostic trace in the file `tld.debug`.

`-Debug` Append a diagnostic trace to the file `tld.debug`. This is useful for tracing several links in the build of a large project.

`-etok` *token-name*

Ensure that regardless of any other hiding flags the token *token-name* is not hidden. A token (or tag) name may be either a string or a unique. A unique is written as `[`$component_1$`.`$component_2$`....` $component_N$`]`. Each component of a unique is a string. A string consists of any sequence of characters, although some special characters must be preceded by a backslash character to stop them being treated specially. These characters are '`\`', '`[`', '`]`' and '`.`'. In addition, the following character sequences are treated the same as they would be in C: '`\n`', '`\r`', '`\t`', '`\0`'. Finally, the sequence '`\x`*NN*' represents the character with code *NN* in hex.

`-etag` *tag-name* Ensure that regardless of other hiding flags the tag *tag-name* is not hidden.

`-hdef` Hide all tags and tokens that have a definition.

`-hdeftok` Hide all tokens that have a definition.

`-hdeftag` Hide all tags that have a definition.

`-htok` token-name

Hide the token named token-name.

`-htag` *tag-name*
: Hide the tag named *tag-name*.

`-hlib` *library*
: Hide all tags and tokens that are not in the library *library*. Libraries are specified in the same way as the `-l` flag, and the same search path is used. Several of these flags may be specified, in which case all tokens and tags not defined in any of the libraries are hidden.

`-L`*directory*
: Add directory to the library search path. The direc- tories are searched in the order specified on the com- mand line.

`-l`*library*
: Use TDF library *library* as a source of definitions. If *library* is an absolute path name, then no searching is performed. If *library* contains a '/' character, then the *library* search path is searched to find the library. If *library* contains no '/' character, then the library search path is searched for a file called *library*`.tl`.

`-nc`
: Don't remove the output file if an error occurs. Normally the file would be removed upon error.

`-nerr`
: Stop attempts at hiding undefined tokens and tags being errors. If an attempt is made to hide an undefined token or tag, it will be left visible (normally this is an error). If the -verbose switch has been specified, all such tokens and tags will be listed.

`-o` *outfile*
: Causes the output to be written to file *outfile* rather than to the default output file `tdf-output.t`.

`-verbose`
: Causes verbose output to be produced. This warns of tokens and tags which have no definition, and of attempts to hide undefined tokens and tags (when used in conjunction with the `-nerr` switch).

LIBRARY CONSTRUCTION

A TDF library is a sequence of named capsules, with an index. The index indicates which tokens and tags are defined by the capsules in the library, and which capsules provide the definitions. When invoked with the `-mc` switch, `tld` produces a library consisting of the TDF capsules specified on the command line. The library is written to the file `tdf-library.tl`, unless the `-o` switch is used.

Switches

Tld accepts the following switches in library construction mode.

`-debug`
: Produce a diagnostic trace in the file `tld.debug`.

`-Debug`	Append a diagnostic trace to the file `tld.debug`. This is useful for tracing several links in the build of a large project.
`-nc`	Don't remove the output file if an error occurs. Normally the file would be removed upon error.
`-o` *outfile*	Causes the output to be written to file *outfile* rather than to the default output file `tdf-library.tl`.
`-verbose`	Causes verbose output to be produced.

LIBRARY CONTENTS

When invoked with the `-mt` switch, `tld` produces a listing of the contents of the library specified on the command line.

Switches

Tld accepts the following switches in library contents mode.

`-verbose`	Causes verbose output to be produced.

LIBRARY EXTRACTION

When invoked with the `-mx` switch, `tld` extracts capsules from the library specified on the command line. The names of the capsules to extract should follow the library name. If no names are specified, all capsules are extracted. If capsule names are specified, they must match exactly the names of the capsules in the library (use the `-mt` mode switch to find out what the exact names are). The capsules are extracted into the current directory, using the basename of the capsule as the output file name. They will overwrite existing files of the same name.

Switches

`tld` accepts the following switches in library extraction mode.

`-nc`	Don't remove the output file if an error occurs. Normally the file would be removed upon error.
`-verbose`	Causes verbose output to be produced.

SEE ALSO

`tcc`(1tdf).