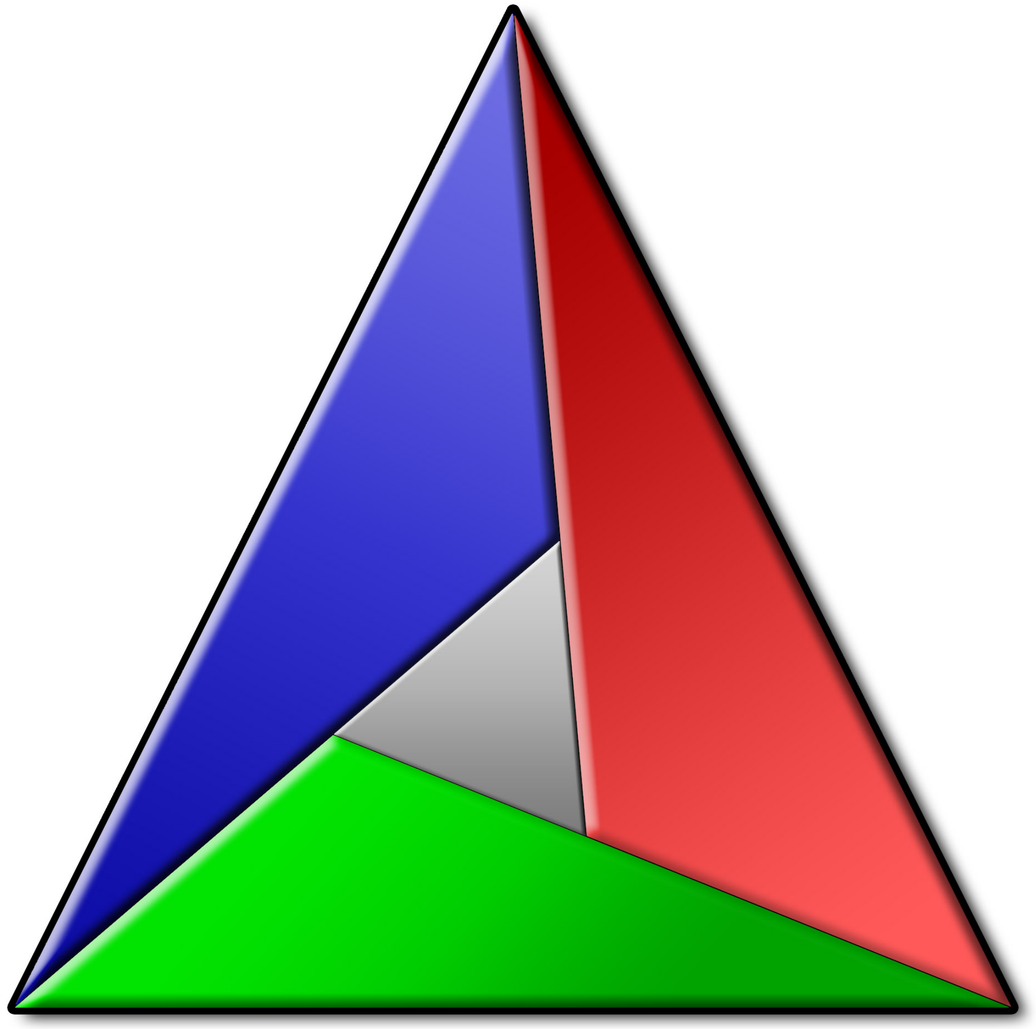
Managing Linux With the Embedded Perspective TX00EX85-3001

1.09.2024



What is GNU Make?
Aside from being awesome

- Make is a utility
  - Typically used to manage the build process of software projects
  - Very often used for GCC, but is not at all limited to that and can be used for any development or software packaging
- Was created in 1976 and released for free in 1980s
- Developed in Bell Labs (Same place that the C programming language was created)

CMake?

- Open-source tools made to build, test, and package software.
- Scripting language written in C++
- Control compilation
  - Creates system independent config files
- Creates Makefiles and workspaces to be used where you want
- CMake is found in a lot of well known applications

CMake?

- Comprised of three CLI tools
  - cmake
    - Creates platform independent build instructions
  - ctest
    - Detects and runs tests
  - cpack
    - Packages software projects into installer packages

CMake?

- Doesn’t build your code
- Generates files for the target platform
  - Make can be used to generate build files for VS
  - Makeflies for linux
  - Xcode files for MacOS

Why CMake?

- Allows configuration for compiler independent builds
- Creates the build files in a separate directory, which enables you to manage multiple builds from the same sources
- Cross-compilation support
- Helps with maintainability by configuring all building, testing, and packaging from a single tool

How does CMake work?

- Configure
  - CMake will parse the CMakeLists.txt file and execute it to configure the build files based on the architecture of the target system, the toolchains to be used, and any dependencies that need to be satisfied
  - CMake then writes the build files based on this

How does CMake work?

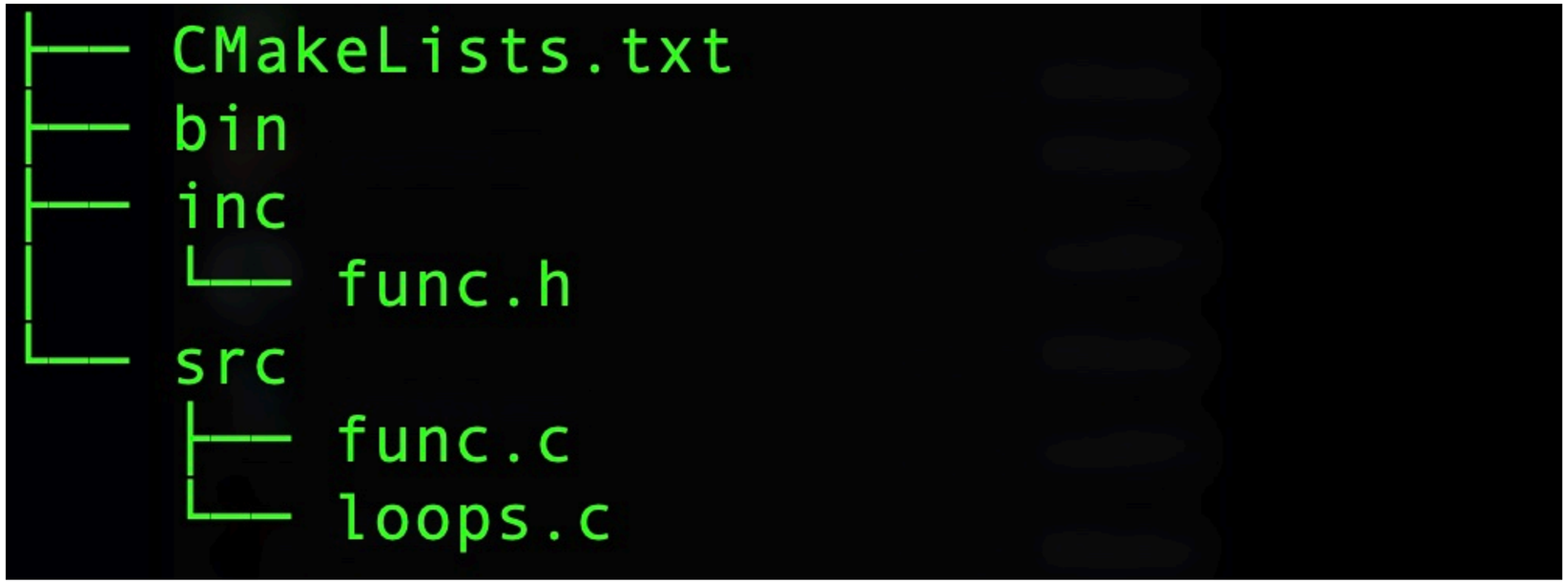
- Build
  - CMake doesn't build so Make is used to do the actual building of the project
  - The build directory that you create will contain the generated binaries, build artifacts, Instructions generated, and the cache
    - CMakeCache.txt will contain the configuration information

CMakeLists.txt

- A project that uses CMake will always include a CMakeLists.txt file
  - Describes the structure of the project
  - Files to be compiled
  - Directories to be included
  - CMake parses the file and generates the desired output

Sample

- Sample project to describe CMake



CMakeLists.txt

- Always starts by setting the minimum CMake version
  - Version number will be the desired CMake version to be used
  - Current CMake version is 3.25.0

```
1 cmake_minimum_required(VERSION <version-number>)
```

```
~ ✔ 16:04:17
cmake --version
cmake version 3.25.0

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

CMakeLists.txt

- The second instruction is setting the project name

```
3 project(<Project name>
4     VERSION <Version Number>
5     DESCRIPTION "<Brief description of your project>"
6     LANGUAGES <Language name i.e. C or CXX>)
```

  - VERSION is the version number of your project software
  - DESCRIPTION is a brief description of your project
  - LANGUAGES is where you give info about the languages of your project

CMakeLists.txt

- Third instruction is for creating your executable

```
 9 add_executable(myApp
10     file1.h
11     file1.c
12     file2.h
13     file2.c
14     main.c)
```

  - The add_executable instruction needs at least two args
  - The fist is the executable name(CMake will handle the format .exe for Windows just the name for MacOS and Linux
  - The second argument are the paths to your source files

CMakeLists.txt

- Then you *can* set the C/C++ standard version that you wish to use

```
16 target_compile_features(<app name> <scope> <C/CPP standard version>)
```

  - You will need to give the executable files name
  - You will also need to set the scope of your application
  - You will need to set a standard version

CMakeLists.txt

- You can also set additional compiler flags flags

```
20 target_compile_options(myApp PRIVATE <list of compiler flags>)
```

  - You will need to give the executable file name
  - You will also need to set the scope of your application
  - Give a list of compiler flags that you would usually give on the CLI

CMake Scope

- CMake targets searches header files from the INCLUDE_DIRECTORIES and INTERFACE_INCLUDE_DIRECTORIES
- The directories included depend on the scope, which is identified by keyword
- PUBLIC is used for all targets and appends directories to INCLUDE_DIRECTORIES and INTERFACE_INCLUDE_DIRECTORIES
- PRIVATE Used for current target only appending directories to INCLUDE_DIRECTORIES
- INTERFACE Will NOT be used for current target, but is available to other targets that depend on the current target, appends to INTERFACE_INCLUDE_DIRECTORIES

CMakeLists.txt

- There are a lot of things that you can add to CMakeLists
  - Multiplatform support by using STREQUAL and if else structure
  - Passing CLI variables into CMake
  - Managing DEBUG or RELEASE builds
  - Managing dependencies
  - Info about CMake can be found here

Using CMake

- Your CMakeLists.txt file should reside in the root of your project

Using CMake

- Create a build directory in the root of the project file
  - Mine is called bin

```
josephhotchkiss@Josephs-MacBook-Pro eLinux % mkdir bin && tree ../eLinux
../eLinux
├── CMakeLists.txt
├── bin
├── inc
│   └── func.h
└── src
    ├── func.c
    └── loops.c
```

Using CMake

- Change director into bin
  - You can see that bin is empty at this time

```
josephhotchkiss@Josephs-MacBook-Pro eLinux % cd bin && tree ../bin
../bin

0 directories, 0 files
josephhotchkiss@Josephs-MacBook-Pro bin %
```

Using CMake

- Now we will source our CMakeLists file
  - cmake ..

```
josephhotchkiss@Josephs-MacBook-Pro bin % cmake ..
-- The C compiler identification is AppleClang 13.0.0.13000029
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
```

```
josephhotchkiss@Josephs-MacBook-Pro bin % tree ../bin
../bin
├── CMakeCache.txt
├── CMakeFiles
│   ├── 3.22.2
│   │   ├── CMakeCCompiler.cmake
│   │   ├── CMakeDetermineCompilerABI_C.bin
│   │   ├── CMakeSystem.cmake
│   │   └── CompilerIdC
│   │       ├── CMakeCCompilerId.c
│   │       ├── a.out
│   │       └── tmp
│   ├── CMakeDirectoryInformation.cmake
│   ├── CMakeOutput.log
│   ├── CMakeTmp
│   ├── Makefile.cmake
│   ├── Makefile2
│   ├── TargetDirectories.txt
│   ├── cmake.check_cache
│   ├── progress.marks
│   └── test.dir
│       ├── DependInfo.cmake
│       ├── build.make
│       ├── cmake_clean.cmake
│       ├── compiler_depend.make
│       ├── compiler_depend.ts
│       ├── depend.make
│       ├── flags.make
│       ├── link.txt
│       ├── progress.make
│       └── src
├── Makefile
└── cmake_install.cmake
```

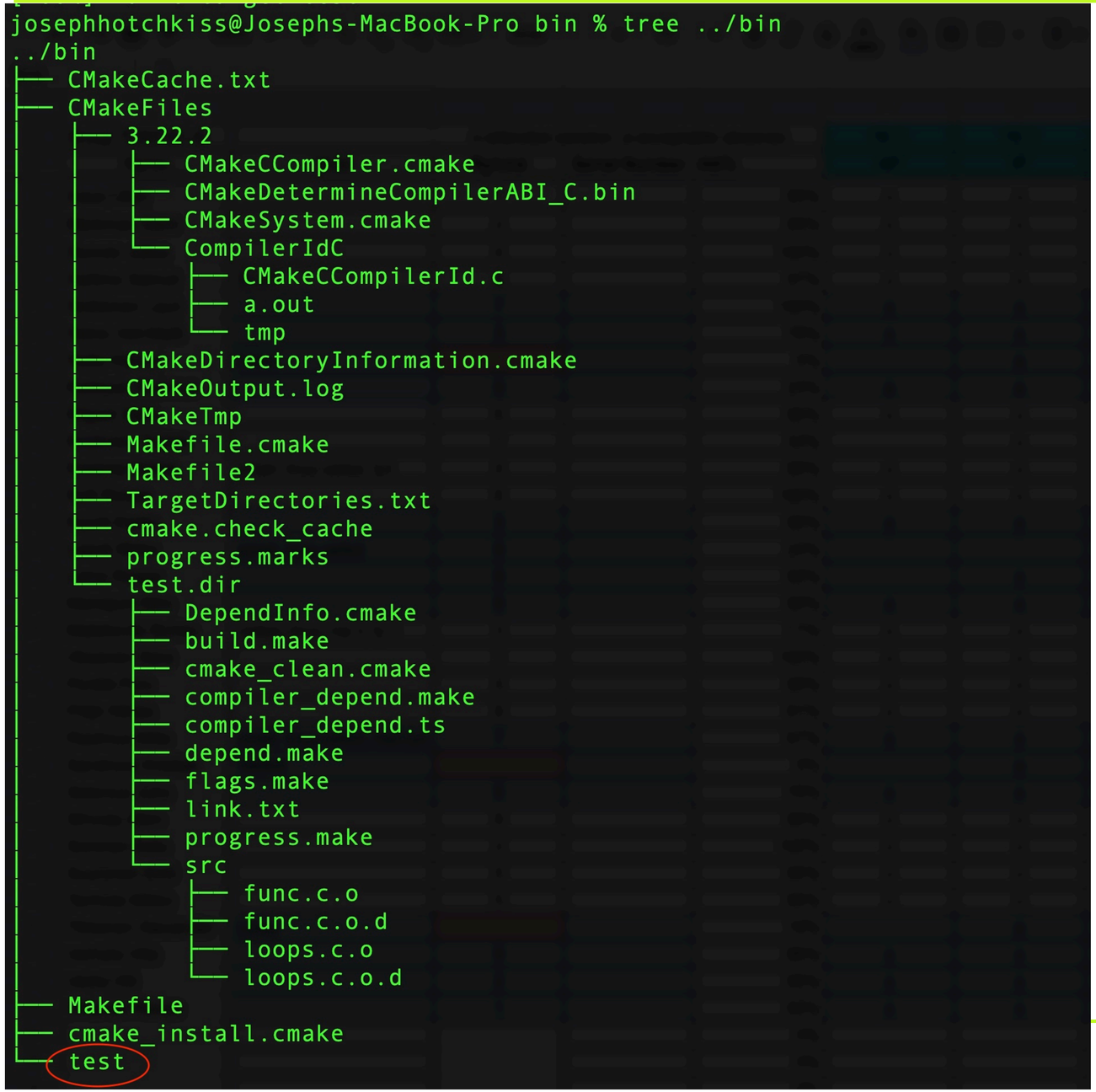
Using CMake

- You will now notice that the bin directory is not empty
  - Please note that we now have a Makefile

Using CMake

- The last step to finish the build is to invoke make
  - If everything has succeeded you have successfully build your project

```
josephhotchkiss@Josephs-MacBook-Pro bin % make
[ 33%] Building C object CMakeFiles/test.dir/src/loops.c.o
[ 66%] Building C object CMakeFiles/test.dir/src/func.c.o
[100%] Linking C executable test
[100%] Built target test
```



Using CMake

- The last step to finish the build is to invoke make
  - If everything has succeeded you have successfully build your project

Using CMake

- Now provided that you didn’t add any new sources or libraries etc. any changes that you make to your code can just be rebuilt by invoking “make”
- If you add new sources etc. you will need to update the CMakeLists.txt file and redo the CMake steps before invoking make