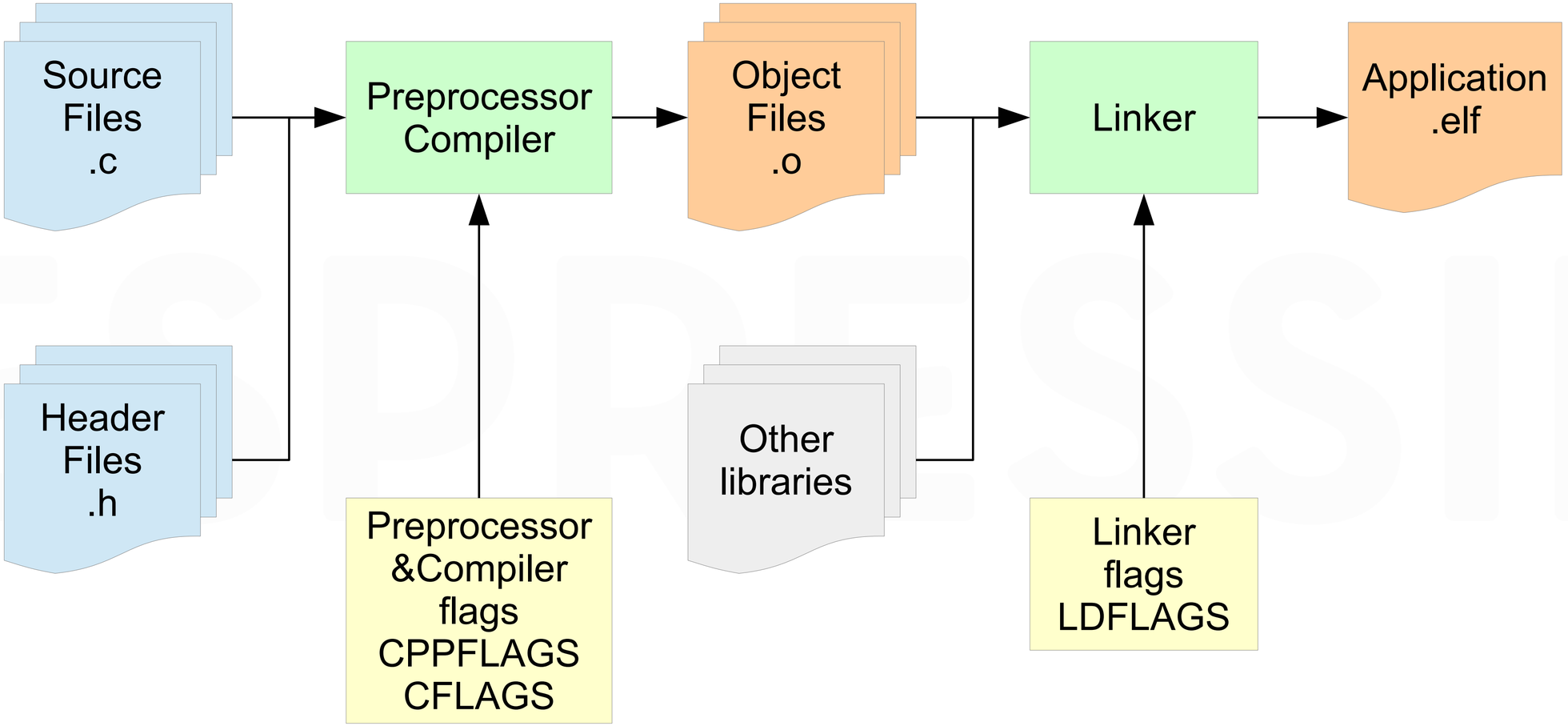
Build Systems

IDF TEAM 2018/09

Overview

- Build systems
  - Build script
  - Make
  - CMake
  - Other build systems
- IDF build system
  - Components
  - Configuration
  - Project

Build process, simple case



Build script

```
#!/bin/bash

source_files="test.c main.c"
# replace all occurrences of '.c' with '.o':
object_files="${source_files%.c/.o}"
program_file=myapp
# 'gcc' can work both as compiler and linker,
# depending on arguments
CC=gcc
LD=gcc

echo "Source files: $source_files"
for c_file in $source_files; do
    # another (better) way to replace extension
    o_file="${c_file%.c}.o"
    echo "Compiling $c_file to $o_file"
    $CC -c -o $o_file $c_file
done

echo "Linking $object_files to $program_file"
$LD -o $program_file $object_files
```

build.sh

```
#include "test.h"

int main(void)
{
    test_func();
}
```

main.c

```
#include <stdio.h>
#include "test.h"

void test_func(void)
{
   printf("Hello from %s\n", __FILE__);
}
```

test.c

```
#pragma once

void test_func(void);
```

test.h

```
$ ./build.sh
Source files: test.c main.c
Compiling test.c to test.o
Compiling main.c to main.o
Linking test.c main.c to myapp
$ ./myapp
Hello from test.c
```

Problems with build script

1. Always compiles all files, even if nothing has changed

   ⇒ need dependency tracking

2. All build steps in one file

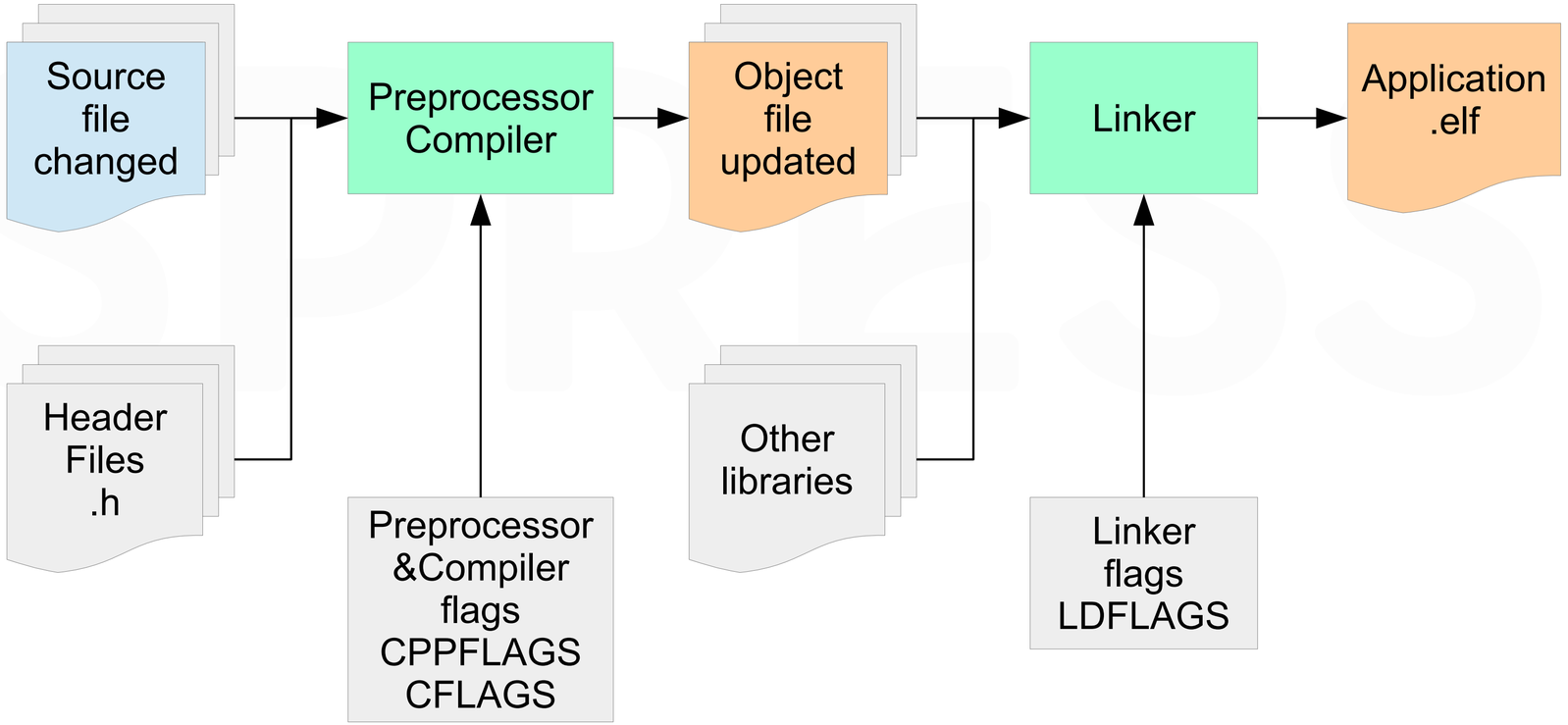
   ⇒ hard to achieve modular design

3. Depends on shell scripting (bash)

   ⇒ needs MSYS or Cygwin to work on Windows

Dependency Tracking

The build system should determine which files have changed, and rebuild only these files.

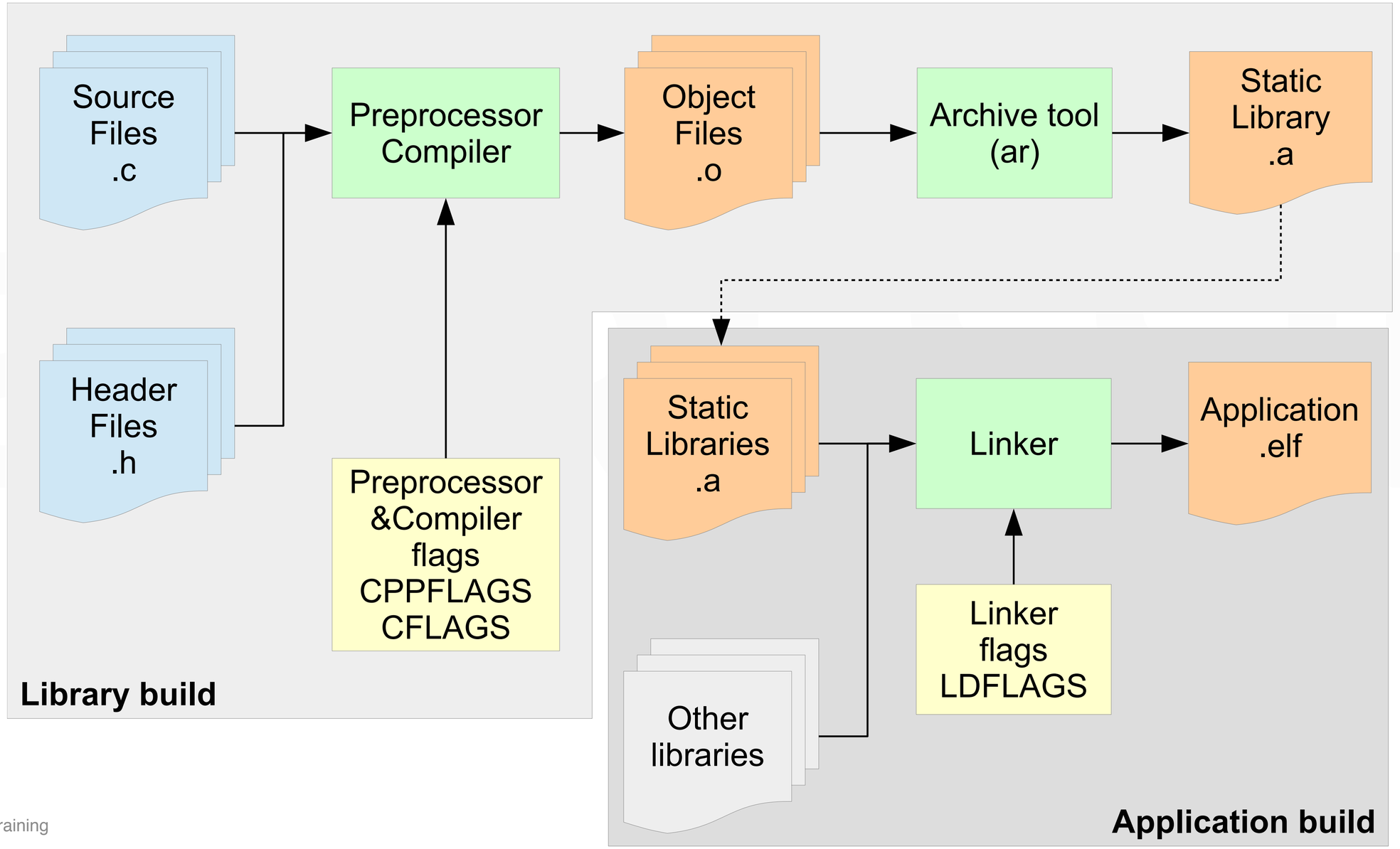


Dependency Tracking (2)

Should also track changes to:

- Header files — rebuild all source files which include that header
- CPPFLAGS, CFLAGS, CXXFLAGS — rebuild all files
- LDFLAGS — re-link the application
- Other libraries (.a) — re-link the application
- Linker script — re-link the application

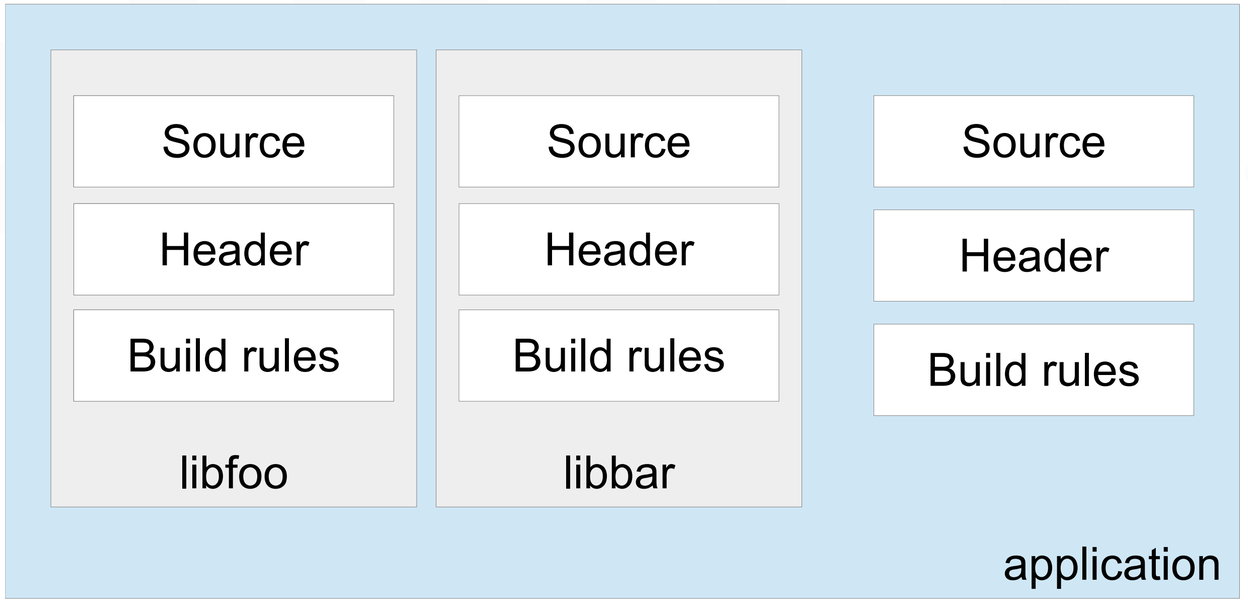
Build process, with libraries



Modular design

Build rules (scripts) for different libraries should be separate from the application build rules (scripts).

…but build settings (build path, compilation flags) should be common.



Introduction to Make

- Build scripts are called *makefiles* (i.e. Makefile)
- Makefiles define targets (usually, files which need to be built) and rules (how to build targets)
- Makefiles can define variables, macros, call built-in functions, include other makefiles.
- The program which executes Makefiles is called ‘make’
- Documentation:
  https://www.gnu.org/software/make/manual/
  http://free-online-ebooks.appspot.com/tools/gnu-make-cn/

Revisiting the simple example

```
#!/bin/bash

source_files="test.c main.c"
# replace all occurrences of '.c' with '.o':
object_files="${source_files%.c/.o}"
program_file=myapp
# 'gcc' can work both as compiler and linker,
# depending on arguments
CC=gcc
LD=gcc

echo "Source files: $source_files"
for c_file in $source_files; do
    # another (better) way to replace extension
    o_file="${c_file%.c}.o"
    echo "Compiling $c_file to $o_file"
    $CC -c -o $o_file $c_file
done

echo "Linking $object_files to $program_file"
$LD -o $program_file $object_files
```

build.sh

```
SOURCE_FILES := test.c main.c
OBJECT_FILES := $(SOURCE_FILES:.c=.o)
PROGRAM_FILE := myapp

$(PROGRAM_FILE): $(OBJECT_FILES)
  $(CC) -o $@ $^
```

Makefile

Simple Makefile explained

```
SOURCE_FILES := test.c main.c
OBJECT_FILES := $(SOURCE_FILES:.c=.o)
PROGRAM_FILE := myapp

$(PROGRAM_FILE): $(OBJECT_FILES)
  $(CC) -o $@ $^

$(OBJECT_FILES): %.o: %.c
  $(CC) -c $< -o $@
```

Variable definitions

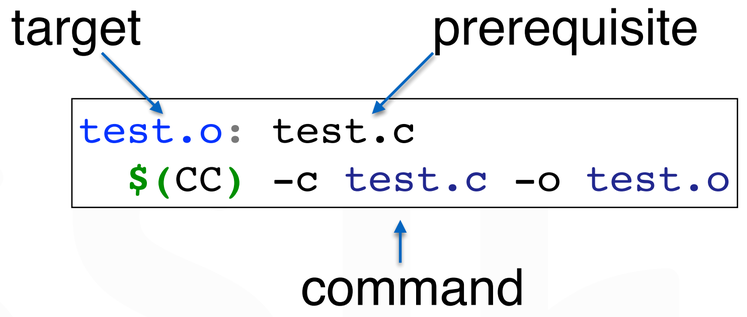
Rule to link the application

Rule to compile the object files (optional)

Targets and rules

- Rule for ‘target’ is run when ‘target’ does not exist (either file or directory)
- If ‘prerequisite’ has changed, ‘target’ will be rebuilt
- If ‘prerequisite’ has not changed, ‘target’ will *not* be rebuilt (even if you ask for it)
- Limitations:
  commands can not set variables, each command runs in a new shell (can’t do *cd*), no multi-line bash constructs (loops, conditions).
- If ‘target’ is not a file or directory, it must be marked ‘.PHONY’

```
targets : prerequisites
  commands
  ...
  ...
```




```
all: build-app run-tests

build-app: $(PROGRAM_FILE)

run-tests: $(PROGRAM_FILE)
  $(PROGRAM_FILE) --test

.PHONY: build-app run-tests all
```

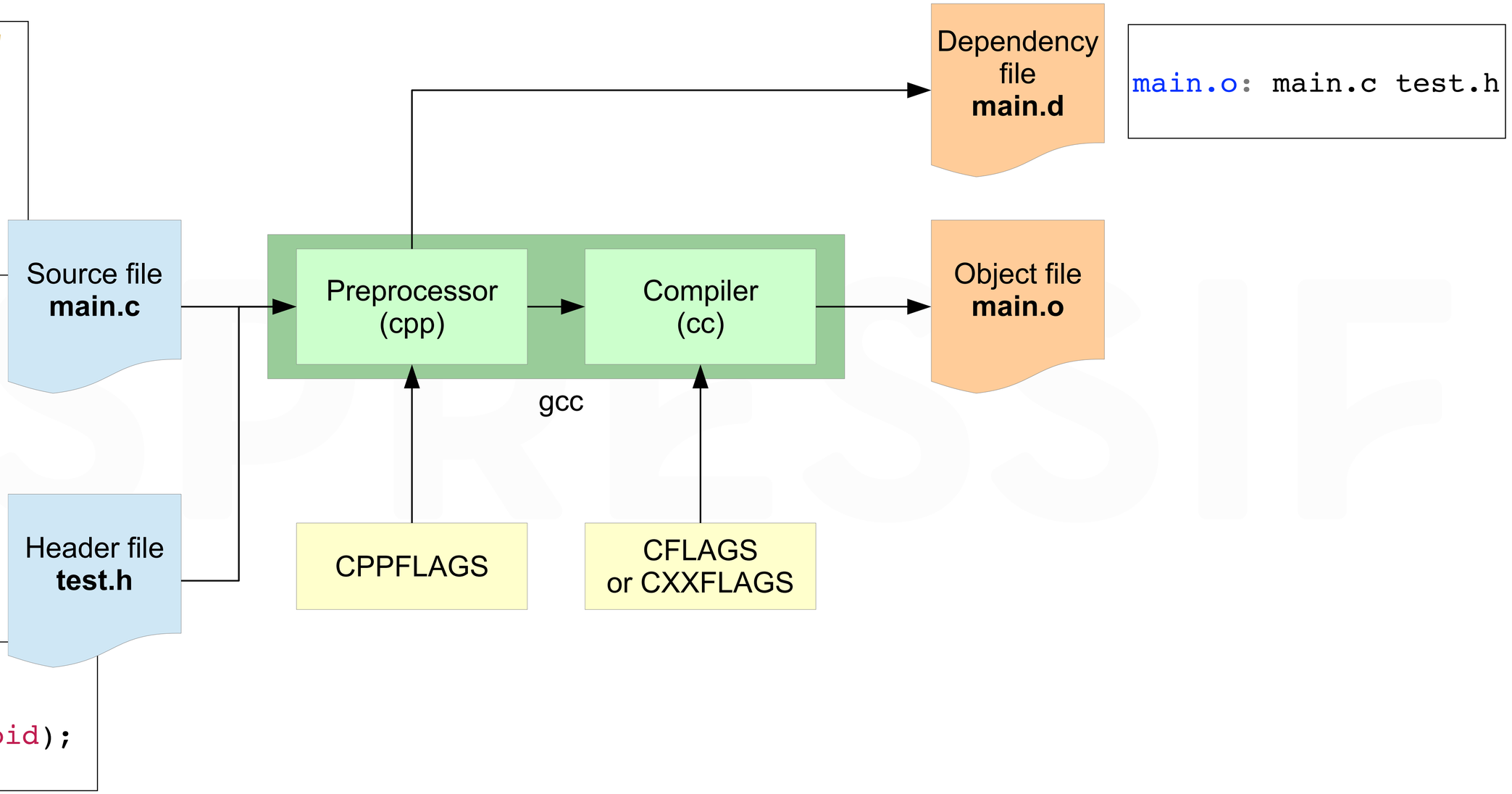
Improving the Makefile

1. Add dependencies on header files
2. Add CPPFLAGS/CFLAGS/LDFLAGS
3. Move output files into build directory
4. Add “all” and “clean” targets

Dependencies on header files

```
#include "test.h"

int main(void)
{
    test_func();
}
```




```
#pragma once

void test_func(void);
```

Adding dependencies to Makefile

```
SOURCE_FILES := test.c main.c
OBJECT_FILES := $(SOURCE_FILES:.c=.o)
DEPENDENCY_FILES := $(SOURCE_FILES:.c=.d)
PROGRAM_FILE := myapp

all: $(PROGRAM_FILE)

$(PROGRAM_FILE): $(OBJECT_FILES)
  $(CC) -o $@ $^

$(OBJECT_FILES): %.o: %.c
  $(CC) -c -MMD $< -o $@

-include $(DEPENDENCY_FILES)

.PHONY: all
```

List of dependency files

Tell GCC to generate dependency files

Tell Make to include dependency files, if they exist

Moving output to build directory

- For each source file, build generates *.o, *.d files
- These files pollute source directory
- Usually it is more convenient to place them into separate directory

Solution:

- Use addprefix function to add build/ to output file names
- Modify pattern rule

Moving output to build directory (2)

```
SOURCE_FILES := test.c main.c
BUILD_DIR := build
OBJECT_FILES := $(addprefix $(BUILD_DIR)/,$(SOURCE_FILES:.c=.o))
DEPENDENCY_FILES := $(addprefix $(BUILD_DIR)/,$(SOURCE_FILES:.c=.d))
PROGRAM_FILE := $(BUILD_DIR)/myapp

all: $(PROGRAM_FILE)

$(PROGRAM_FILE): $(OBJECT_FILES) | $(BUILD_DIR)
  $(CC) -o $@ $^

$(OBJECT_FILES): $(BUILD_DIR)/%.o: %.c | $(BUILD_DIR)
  $(CC) -c -MMD $< -o $@

$(BUILD_DIR):
  mkdir -p $@

-include $(DEPENDENCY_FILES)

.PHONY: all
```

Adding FLAGS and cleaning up

```
SOURCE_FILES := test.c main.c
BUILD_DIR := build
OBJECT_FILES := $(addprefix $(BUILD_DIR)/,$(SOURCE_FILES:.c=.o))
DEPENDENCY_FILES := $(addprefix $(BUILD_DIR)/,$(SOURCE_FILES:.c=.d))
PROGRAM_FILE := $(BUILD_DIR)/myapp

CFLAGS ?= -Og -Wall -Werror
CPPFLAGS ?=
LDFLAGS ?= -Og

all: $(PROGRAM_FILE)

$(PROGRAM_FILE): $(OBJECT_FILES) | $(BUILD_DIR)
	$(CC) $(LDFLAGS) -o $@ $^

$(OBJECT_FILES): $(BUILD_DIR)/%.o: %.c | $(BUILD_DIR)
	$(CC) $(CPPFLAGS) $(CFLAGS) -c -MMD $< -o $@

$(BUILD_DIR):
	mkdir -p $@

-include $(DEPENDENCY_FILES)

clean:
	rm -f $(PROGRAM_FILE) $(OBJECT_FILES) $(DEPENDENCY_FILES)

.PHONY: all clean
```

Using make

Normal build

Program runs

Nothing changed

Only test.c is rebuilt

All files which include test.h are rebuilt

Clean build output

Set some build flag

```
$ make
mkdir -p build
cc  -Og -Wall -Werror -c -MMD test.c -o build/test.o
cc  -Og -Wall -Werror -c -MMD main.c -o build/main.o
cc -Og -o build/myapp build/test.o build/main.o
$ build/myapp
Hello from test.c
$ make
make: Nothing to be done for `all'.
$ touch test.c
$ make
cc  -Og -Wall -Werror -c -MMD test.c -o build/test.o
cc -Og -o build/myapp build/test.o build/main.o
$ touch test.h
$ make
cc  -Og -Wall -Werror -c -MMD test.c -o build/test.o
cc  -Og -Wall -Werror -c -MMD main.c -o build/main.o
cc -Og -o build/myapp build/test.o build/main.o
$ make clean
rm -f build/myapp build/test.o build/main.o build/test.d build/main.d
$ make CFLAGS=-O2
cc  -O2 -c -MMD test.c -o build/test.o
cc  -O2 -c -MMD main.c -o build/main.o
cc -Og -o build/myapp build/test.o build/main.o
$
```

Let's try the same in CMake

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.5)
project(myapp)
add_executable(myapp test.c main.c)
```

```
$ mkdir build
$ cd build
$ cmake ..
-- The C compiler identification is AppleClang 9.1.0.9020039
-- The CXX compiler identification is AppleClang 9.1.0.9020039
-- Check for working C compiler: /Library/Developer/CommandLine
-- Check for working C compiler: /Library/Developer/CommandLine
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
(some lines removed)
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/ivan/e/tr/build_sys
$ make
Scanning dependencies of target myapp
[ 33%] Building C object CMakeFiles/myapp.dir/test.c.o
[ 66%] Building C object CMakeFiles/myapp.dir/main.c.o
[100%] Linking C executable myapp
[100%] Built target myapp
```

CMake

- CMake is designed to build C or C++ applications ⇒

  knows about dependencies out of the box

- CMake is a build system generator ⇒

  generates Makefiles, Eclipse/MSVC/Xcode projects and many more
  (try `cmake .. -G "Eclipse CDT4 - Unix Makefiles”`)

- Unlike GNU Make, can work on Windows without MSYS

- Tutorial: https://cmake.org/cmake-tutorial/
  https://www.zybuluo.com/khan-lau/note/254724

Other build systems

Ninja: cross-platform build system, optimised for speed. CMake can generate Ninja build scripts.
https://ninja-build.org/

SCons: build system written in Python. Build scripts are Python programs. Can build programs in C, C++, Java, Fortran, TeX, and other languages.

Meson: another build system, optimised for speed and simplicity.

ESP-IDF Build System

- **Why need a custom build system?**
  - Components
  - Configuration system (Kconfig)
  - Simplify makefiles for common use cases
  - Hide rough edges in cross-platform support
  - flash/monitor
- **First version written in Make**
  - Cross-platform, many developers were familiar with it
- **Recent rewrite in CMake**
  - Better Windows support, improved build times (esp. with ninja), better IDE support (Eclipse, VS Code)

ESP-IDF Build System Documentation

-Make based build system:
https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/build-system.html

-CMake based build system:
https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/build-system-cmake.html

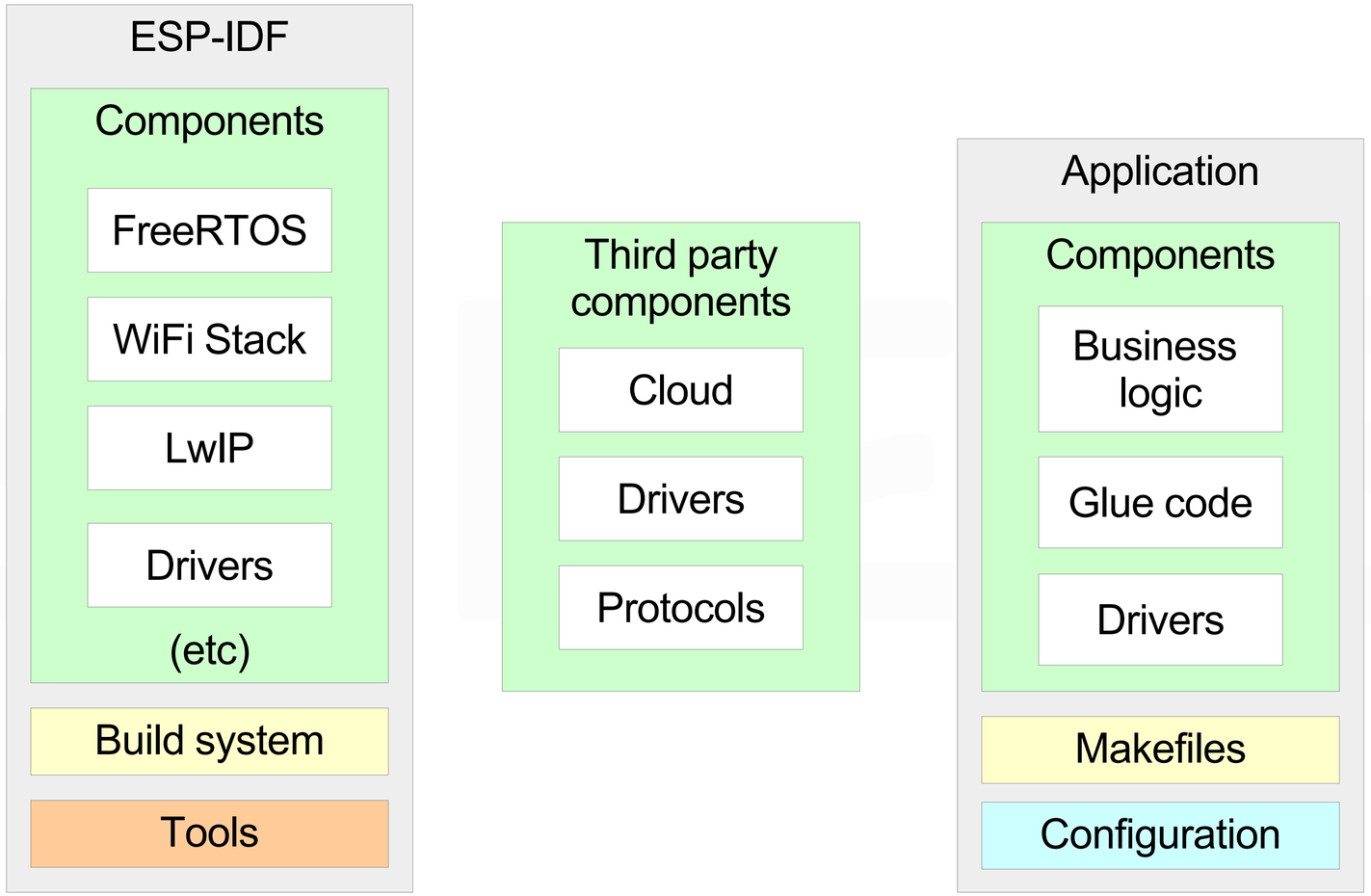
Future of ESP-IDF Build System

Time	Make based build system	CMake based build system
ESP-IDF v3.0	The only build system	feature/cmake preview branch
ESP-IDF v3.1, Master branch (2018/09/14)	Main build system, intended for most users	Preview version. Some features missing.
ESP-IDF v3.2	Main build system, intended for most users	Preview version. Some features (flash encryption, secure boot) are still missing.
ESP-IDF v4.0	Still available, not recommended for new projects	Main build system, intended for most users. All features available.
ESP-IDF v5.0	Removed	The only build system

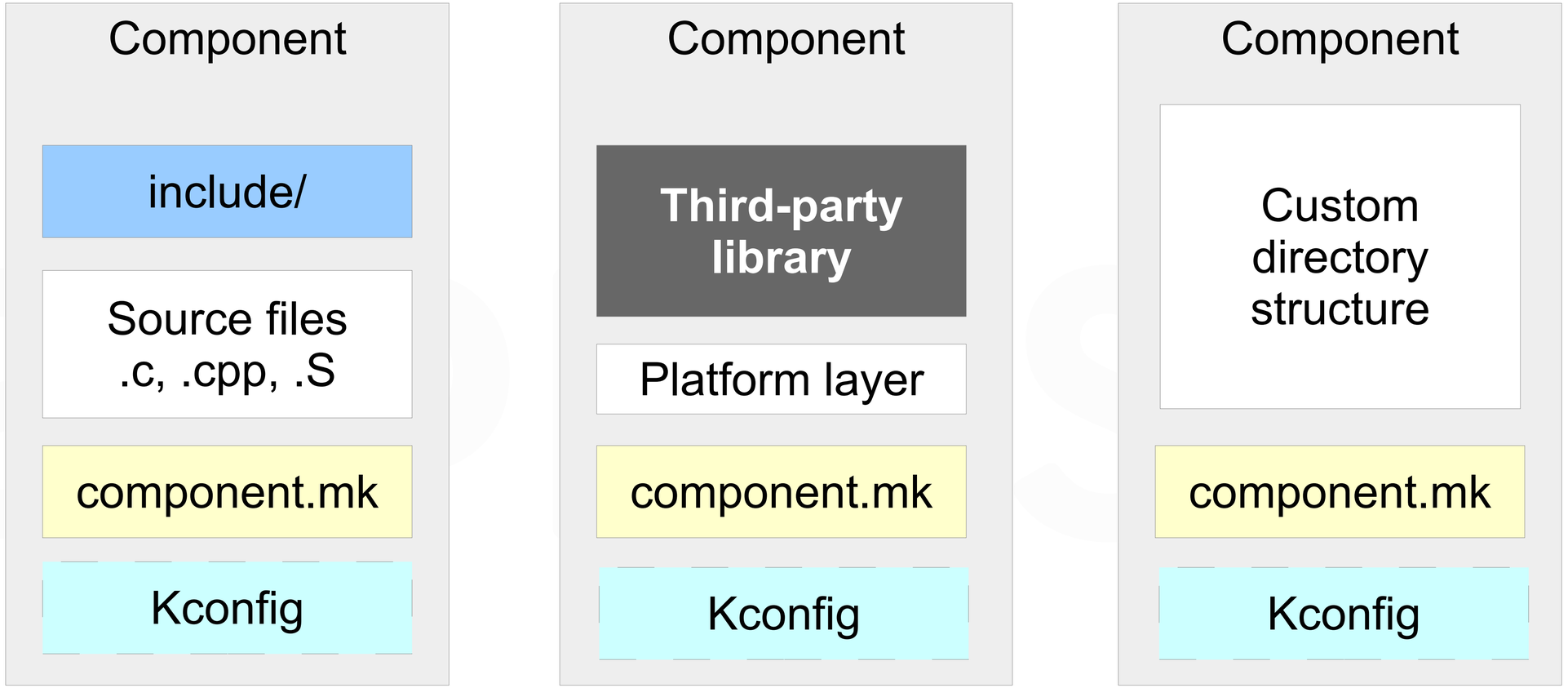
And what this means for Espressif developers…

Time	Make based build system	CMake based build system	Espressif developers
ESP-IDF v3.0	The only build system	feature/cmake preview branch	Support 1 build system
ESP-IDF v3.1, Master branch (2018/09/14)	Main build system, intended for most users	Preview version. Some features missing.	Support 2 build systems
ESP-IDF v3.2	Main build system, intended for most users	Preview version. Some features (flash encryption, secure boot) are still missing.	Support 2 build systems
ESP-IDF v4.0	Still available, not recommended for new projects	Main build system, intended for most users. All features available.	Support 2 build systems
ESP-IDF v5.0	Removed	The only build system	Support 1 build system

ESP-IDF: Projects and Components



ESP-IDF: Component Layout

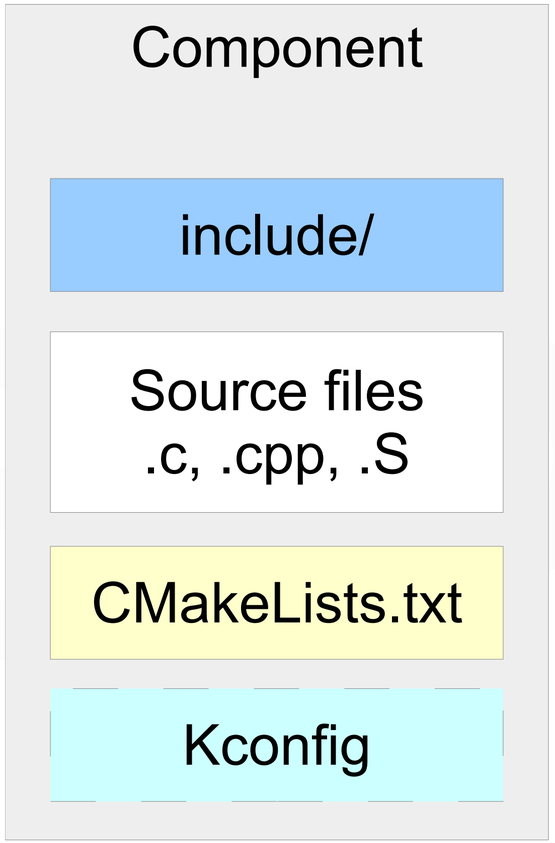


Typical component

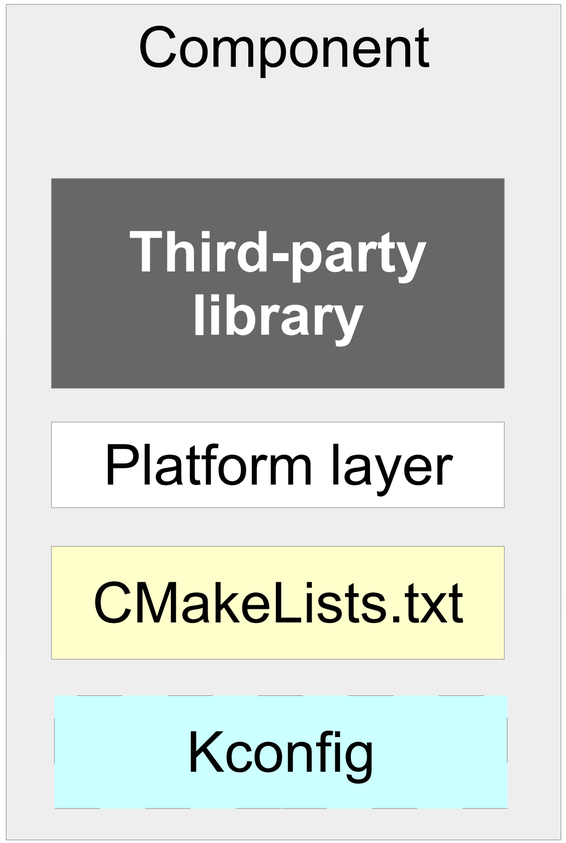
Wrapper component

Fully custom component

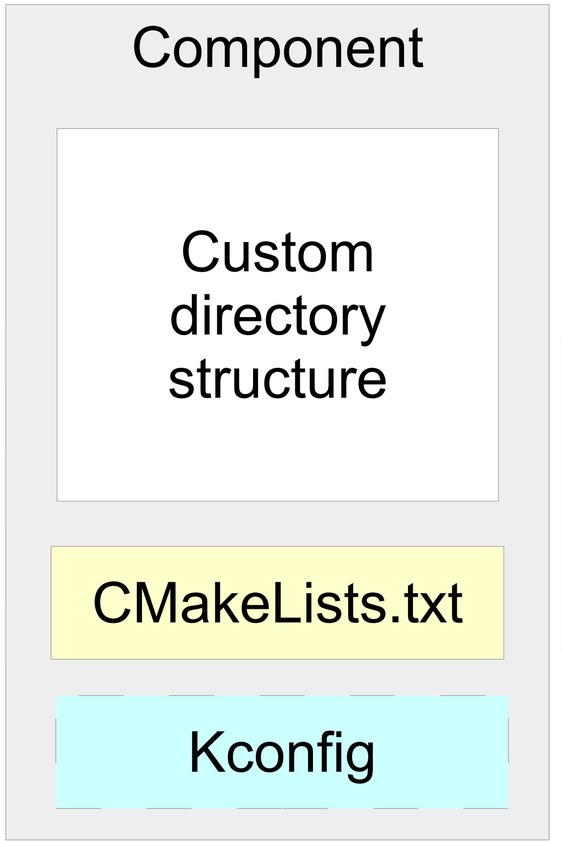
ESP-IDF: Component Layout, CMake



Typical component

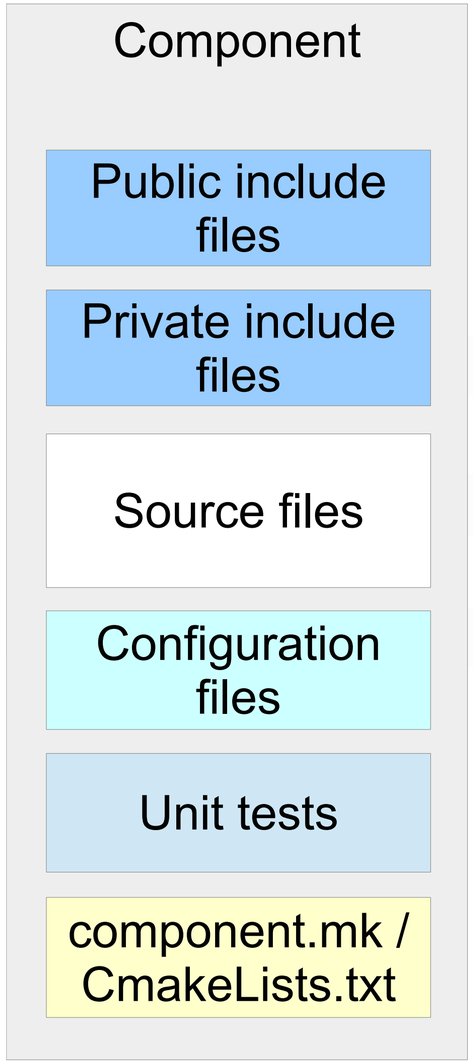


Wrapper component



Fully custom component

ESP-IDF: Component Layout (2)



Recipes for Component Makefiles

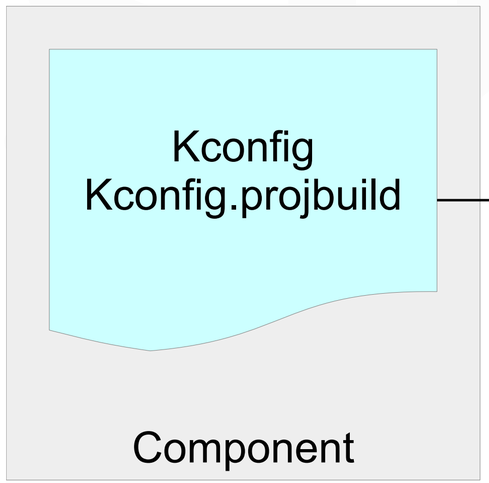
Aspect	CMake based build system (CMakeLists.txt)	Make based build system (component.mk)
Default behaviour	*nothing is compiled*	COMPONENT_SRCDIRS := . COMPONENT_INCLUDEDIRS := include
Set source files	Set source files: set(COMPONENT_SRCS “file.c")	Set source directories: COMPONENT_SRCDIRS = src
Set public include directories	set(COMPONENT_ADD_INCLUDEDIRS “include”)	COMPONENT_ADD_INCLUDEDIRS := include
Set private include directories	set(COMPONENT_PRIV_INCLUDEDIRS “private_include”)	COMPONENT_PRIV_INCLUDEDIRS := private_include
Register component	register_component()	*registered automatically*

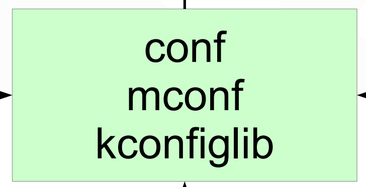
Recipes for Component Makefiles (2)

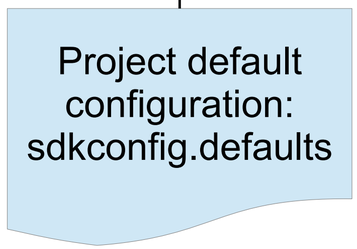
Aspect	CMake based build system (CMakeLists.txt)	Make based build system (component.mk)
Set CFLAGS for the whole component	component_compile_options(-fsome-flag)	CFLAGS += -fsome-flag
Set CFLAGS for source file	set_source_files_properties(myfile.c PROPERTIES COMPILE_FLAGS -fsome-flag)	myfile.o: CFLAGS += -fsome-flag
Set dependencies	set(COMPONENT_REQUIRES some_component)	*no support for dependencies*
Embed files	set(COMPONENT_EMBED_FILES file.bin) set(COMPONENT_EMBED_TXTFILES root.pem)	COMPONENT_EMBED_FILES := file.bin COMPONENT_EMBED_TXTFILES := root.pem

ESP-IDF Configuration System

```
config FOO_DEBUG_ENABLE
    bool "Enable debugging feature"
    default "n"
    help
      Enable debugging feature for
      component Foo
```








```
CONFIG_FOO_ENABLE=y
```

Makefile fragment

```
CONFIG_XXX=y
CONFIG_FOO_ENABLE=y
CONFIG_YYY=42
```

C, C++ header file

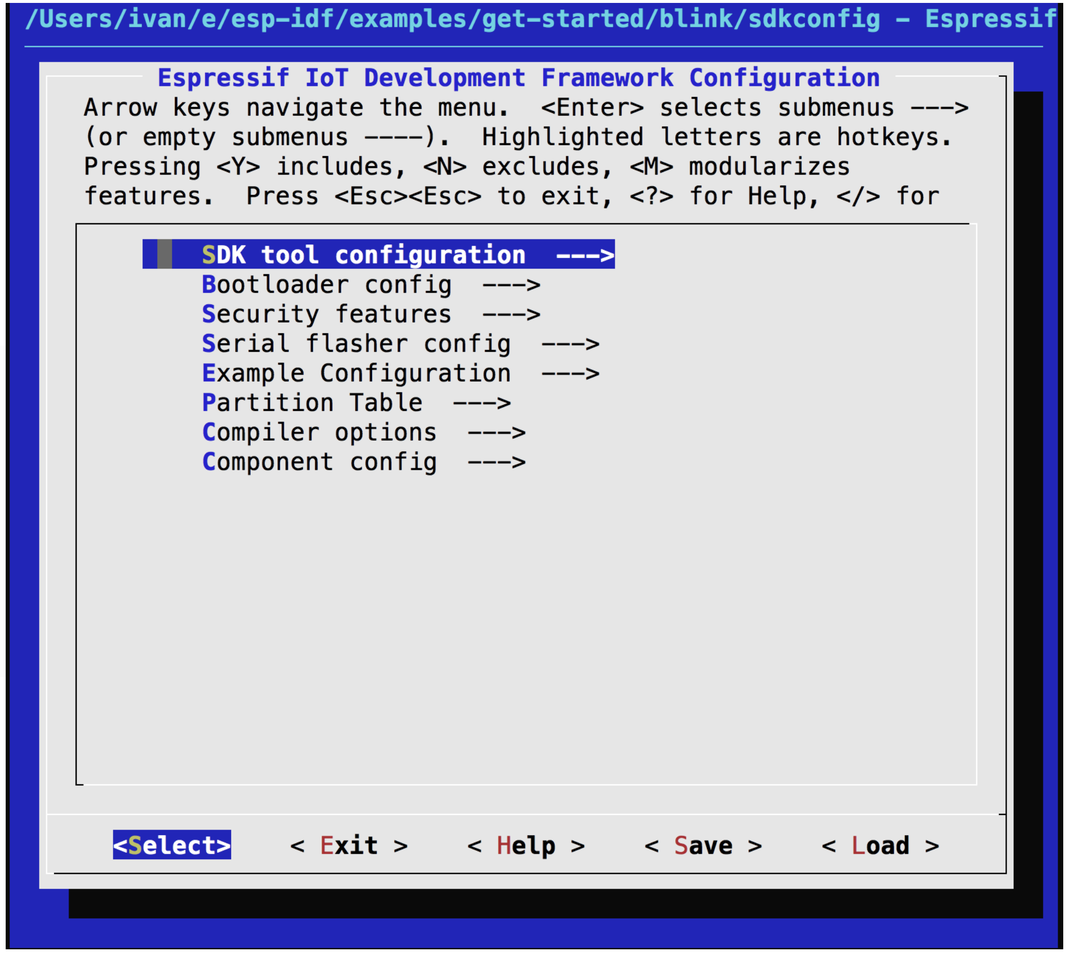
```
#define CONFIG_XXX 1
#define CONFIG_FOO_ENABLE 1
#define CONFIG_YYY 42
```

Project configuration file

```
CONFIG_XXX=y
CONFIG_FOO_ENABLE=y
CONFIG_YYY=42
```

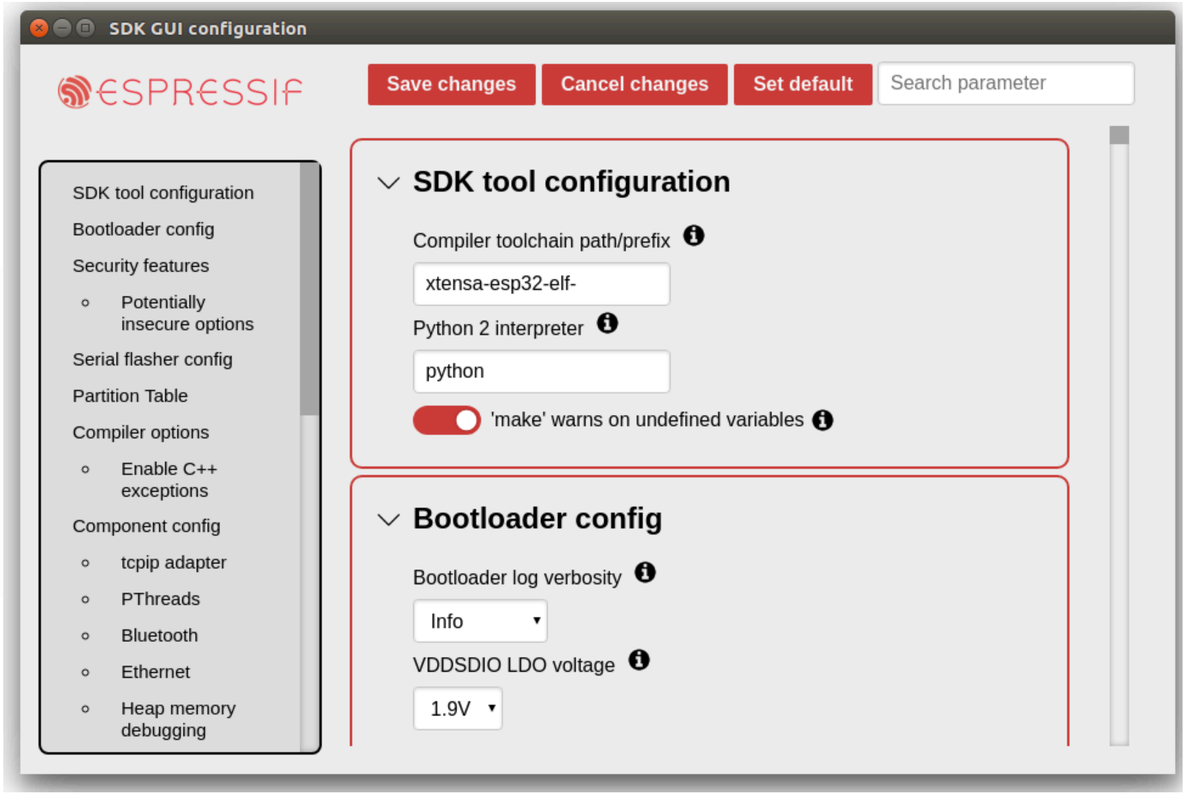
KConfig language documentation:
https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

ESP-IDF Configuration Tools



mconf

(`make menuconfig`, `idf.py menuconfig`)



(WIP) guiconfig

ESP-IDF Component Unit Tests

```
components/
----foo/
    +---include/
    |   +---foo.h
    +---foo.c
    +---test/
    |   +---test_foo.c
    |   +---component.mk (CMakeLists.txt)
    +---component.mk (CMakeLists.txt)
```

test sub-component

ESP-IDF Project Layout

```
project/
+---Makefile / CMakeLists.txt
+---sdkconfig
+---sdkconfig.defaults
+---main/
|   +---component.mk / CMakeLists.txt
|   +---main.c
+---components/
    +---driver-foo/
    |       …
    +---library-bar/
    |       …
```

Recipes for Project Makefiles

Aspect	CMake based build system (CMakeLists.txt)	Make based build system (Makefile)
Minimal project	cmake_minimum_required(VERSION 3.5) include($ENV{IDF_PATH}/tools/cmake/project.cmake) project(my_project)	PROJECT_NAME := my_project include $(IDF_PATH)/make/project.mk
Add custom component directories	set(EXTRA_COMPONENT_DIRS extra_components)	EXTRA_COMPONENT_DIRS := extra_components
Reduce the list of components	set(COMPONENTS mqtt) (required components are added automatically)	COMPONENTS := *<mandatory components>* mqtt (required components have to be listed manually) or EXCLUDE_COMPONENTS := *<components which you don't need>*

Extra Features of ESP-IDF build system

- **Set default port and baud rate:**

```
export ESPPORT=/dev/ttyUSB0
export ESPBAUD=2000000
make flash monitor
idf.py flash monitor
# CMake:
idf.py -p /dev/ttyUSB2 flash monitor
```

- **Check memory usage:**

  make size
  make size-components
  make size-files
  (For CMake: idf.py size, idf.py size-components, idf.py size-files)

- **Erase flash**

  make erase_flash
  make erase_ota

- **Help!**

  idf.py --help
  make help

Q&A