Introduction to Build Systems and CMake

DAY 3 : CMake for Embedded Systems

Sponsored by

Webinar Logistics

- Turn on your system sound to hear the streaming presentation.
- If you have technical problems, click "Help" or submit a question asking for assistance.
- Participate in 'Group Chat' by maximizing the chat widget in your dock.

01

Review: The Problem

The Problem

There are several problems that teams are facing:

- Managing multiple build configurations
- Slow builds
- Software quality issues
- Inability to use modern techniques like DevOps, Simulation, TDD, etc, effectively
- Productivity issues (time to market, product quality)

The Solution

A carefully designed CMake build system will:

- Simplify build configurations with better dependency management
- Allow for faster, cross-platform builds
- Enable consistency across different development environments
- Unlock modern development processes and tools like DevOps, Simulation, and TDD
- Increase productivity

THE SPEAKER

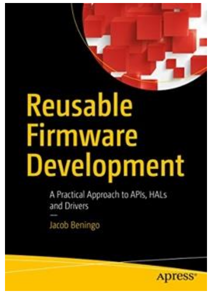
Jacob Beningo

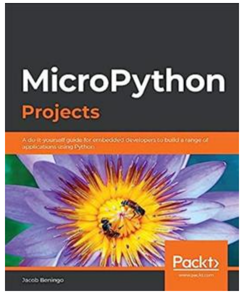
Jacob@beningo.com

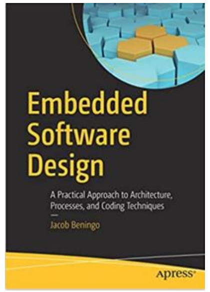
Beningo Embedded Group – CEO / Founder

Focus: Embedded Software Consulting and Training

Help teams deliver higher-quality embedded software faster. We specialize in creating and promoting embedded software excellence in businesses around the world.







Blogs for:

- DesignNews.com
- Embedded.com
- EmbeddedRelated.com
- MLRelated.com

Visit **www.beningo.com** to learn more

The Plan

Transform Your Build Process: Streamline, Modernize, and Boost Productivity with CMake

Step 1

Learn the Technology

Step 2

Design the Solution

Step 3

Adopt Modern Practices

02

Toolchain Files

Toolchain Files - Introduction

A CMake toolchain file is a script used by CMake to define the compilation environment, particularly for cross-compilation scenarios. It allows you to

- specify the compiler
- linker,
- and various other tools and flags

that CMake should use when generating build files. Toolchain files are essential when you are building software for a different platform than the one you are working on, such as when targeting an embedded system from a desktop environment.

Toolchain Files – The Structure

A typical CMake toolchain file is a plain text file with the .cmake extension, and it contains a series of commands that configure the necessary tools and flags

```cmake
# Set the target system name, e.g., 'Linux', 'Windows', 'Generic', etc.
set(CMAKE_SYSTEM_NAME Generic)

# Specify the cross compiler to use
set(CMAKE_C_COMPILER /path/to/your/compiler/arm-none-eabi-gcc)
set(CMAKE_CXX_COMPILER /path/to/your/compiler/arm-none-eabi-g++)

# Set the tool for archiving libraries
set(CMAKE_AR /path/to/your/compiler/arm-none-eabi-ar)

# Set the tool for linking binaries
set(CMAKE_LINKER /path/to/your/compiler/arm-none-eabi-ld)

# Set the tool for the assembler
set(CMAKE_ASM_COMPILER /path/to/your/compiler/arm-none-eabi-as)

# Define any required compiler flags, such as for the CPU architecture
set(CMAKE_C_FLAGS "-mcpu=cortex-m4 -mthumb")
set(CMAKE_CXX_FLAGS "-mcpu=cortex-m4 -mthumb")
```

Toolchain Files - Compilation

Must specify in our command!

```
cmake -DCMAKE_TOOLCHAIN_FILE=$TOOLCHAIN_FILE -G Ninja -B $BUILD_DIR -S . -DCMAKE_BUILD_TYPE=$BUILD_TYPE
ninja -C $BUILD_DIR
```

Audience POLL Question

What is a toolchain file used for?

a) To manage source code versioning in a project
b) To define the compilation environment, particularly for cross-compiling to a different platform
c) To automate the testing of code during the build process
d) To configure the user interface settings in a development environment

03

Host Toolchains

Host Toolchain Files – What are they for?

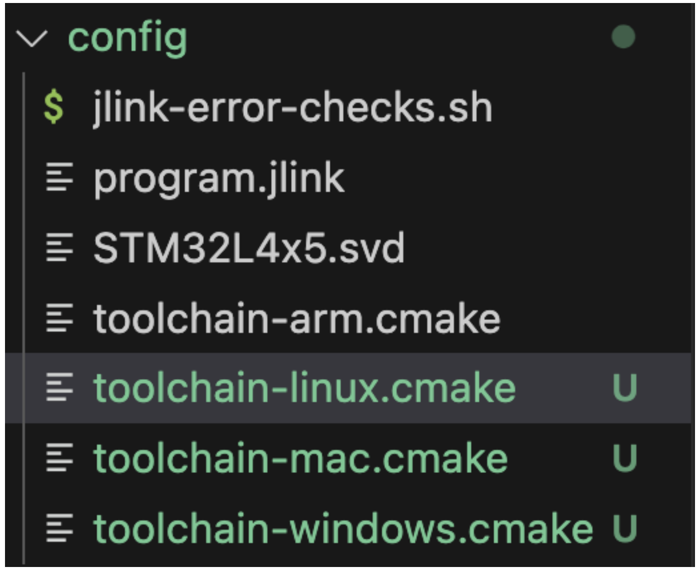
A Host Toolchain file is a configuration file that defines the tools, compilers, and libraries used when building software on the host machine (the machine where the build is happening).

It sets up the environment to ensure consistent builds across different machines by specifying which compiler, linker, and other tools should be used.

Host Toolchain Files – Why do we need them?

- Embedded projects often require specific versions of compilers, linkers, and other tools that might not be the default on every developer's machine. A host toolchain file ensures these requirements are met consistently.

- For Example: Compiling an RTOS



Host Toolchain Files – Threading Example

```
# Set the system name (e.g., ARM Cortex-M)
set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_SYSTEM_PROCESSOR arm)

# Set the cross compiler
set(CMAKE_C_COMPILER arm-none-eabi-gcc)
set(CMAKE_CXX_COMPILER arm-none-eabi-g++)

# Specify the path to FreeRTOS source
set(FREERTOS_PATH /path/to/freertos/source)

# Add compiler flags specific to the embedded platform
set(CMAKE_C_FLAGS "-mcpu=cortex-m4 -mthumb -O2 -
ffreestanding -fno-builtin")
set(CMAKE_CXX_FLAGS "-mcpu=cortex-m4 -mthumb -O2 -
ffreestanding -fno-builtin")

# Include FreeRTOS in the build
include_directories(${FREERTOS_PATH}/include)
```

```
# Set the system name (Linux)
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR x86_64)

# Set the compiler to GCC
set(CMAKE_C_COMPILER /usr/bin/gcc)
set(CMAKE_CXX_COMPILER /usr/bin/g++)

# Specify the path to FreeRTOS source for Linux
set(FREERTOS_PATH /path/to/freertos/source)

# Add necessary flags for building on Linux
set(CMAKE_C_FLAGS "-O2 -Wall")
set(CMAKE_CXX_FLAGS "-O2 -Wall")

# Include FreeRTOS in the build
include_directories(${FREERTOS_PATH}/include)

# Link with the pthread library (often required for RTOS-like
behavior on Linux)
set(CMAKE_EXE_LINKER_FLAGS "-lpthread")
```

```
# Set the system name (Windows)
set(CMAKE_SYSTEM_NAME Windows)
set(CMAKE_SYSTEM_PROCESSOR x86_64)

# Set the compiler to MinGW GCC
set(CMAKE_C_COMPILER C:/mingw/bin/gcc.exe)
set(CMAKE_CXX_COMPILER C:/mingw/bin/g++.exe)

# Specify the path to FreeRTOS source for Windows
set(FREERTOS_PATH C:/path/to/freertos/source)

# Add necessary flags for building on Windows
set(CMAKE_C_FLAGS "-O2 -Wall")
set(CMAKE_CXX_FLAGS "-O2 -Wall")

# Include FreeRTOS in the build
include_directories(${FREERTOS_PATH}/include)

# Link with the Windows threading library (if needed)
set(CMAKE_EXE_LINKER_FLAGS "-lwinpthread")
```

Audience POLL Question

What is the most important reason to use a host toolchain file in embedded software development?

a) To ensure consistent build environments across different development machines
b) To simplify cross-compilation for multiple target platforms
c) To automate the inclusion of third-party libraries and dependencies
d) To optimize build times by using custom compiler and linker settings

04

Target Toolchain Files

Target Toolchain Files – What are they for?

A Target Toolchain file is a configuration file that defines the tools, compilers, and libraries used when cross-compiling software on a host machine (the machine where the build is happening) for a different target.

It sets up the environment to ensure consistent build for the target architecture.

Target Toolchain Files – An Example

```
# By setting CMAKE_SYSTEM_NAME to "Generic," you indicate to CMake that it should avoid platform-specific
# configurations and try to generate a more generic build system that can be used across different
# environments. This is often done in cross-compilation scenarios or when building code that is intended to
# be platform-independent.
set(CMAKE_SYSTEM_NAME Generic)

# Set the toolchain base path
set(TOOLCHAIN_BASE_PATH "/home/dev/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-eabi/bin/")

# Specify the cross compiler and associated tools
set(CMAKE_C_COMPILER "${TOOLCHAIN_BASE_PATH}arm-none-eabi-gcc")
set(CMAKE_CXX_COMPILER "${TOOLCHAIN_BASE_PATH}arm-none-eabi-g++")
set(CMAKE_ASM_COMPILER "${TOOLCHAIN_BASE_PATH}arm-none-eabi-gcc")
set(OBJCOPY_PATH "${TOOLCHAIN_BASE_PATH}arm-none-eabi-objcopy")
set(SIZE_TOOL "${TOOLCHAIN_BASE_PATH}arm-none-eabi-size")
```

Target Toolchain Files – An Example

```
# MCU specific option flags
# We use set to create a list of flags that we want to pass to the compiler. It is a list of strings.
# This is a convenient and configurable method. The flags we are passing to the compiler include:
#
# -mcpu=cortex-m4:    Specifies the target CPU architecture as Cortex-M4.
# -mthumb:            Indicates that the code should be compiled for the Thumb instruction set, which is
#                     commonly used in ARM-based microcontrollers for code size optimization.      You, 3 months ago •
# -mfpu=fpv4-sp-d16:  Specifies the FPU (Floating-Point Unit) type for Cortex-M4, in this case, "fpv4-sp-d16"
#                     stands for a single-precision FPU with 16 double-precision registers.
# -mfloat-abi=hard:   Specifies that the code should use the "hard" floating-point ABI (Application Binary Interface),
#                     which means that floating-point calculations should be performed using hardware instructions
#                     (as opposed to software emulation).
# --specs=nano.specs: This flag is specific to some ARM toolchains (like the GCC ARM toolchain) and is used to
#                     specify linker options for using the Nano Standard C Library, which is a minimalistic
#                     version of the C library optimized for embedded systems with limited resources.
set(MCU_FLAGS
    -mcpu=cortex-m4
    -mthumb
    -mfpu=fpv4-sp-d16
    -mfloat-abi=hard
    -specs=nano.specs
)
```

Continuing Education Center

BENINGO EMBEDDED GROUP

DigiKey

Target Toolchain Files – An Example

```
# Define additional compiler symbols (-D)
#
# USE_HAL_DRIVER:     Symbol used to tell STM32 library we are using the HAL drivers
# STM32L475xx:        Symbol specifying the MCU target family
# __FPU_PRESENT       Tell the ST libraries that the FPU is present
add_compile_definitions(
    USE_HAL_DRIVER
    STM32L475xx
    __FPU_PRESENT=1U
)
```

```
# g:                   Generate debug information
# O2:                  Optimize code for speed
# Wall:                Enables a compiler warning messages that catch common programming errors
# Wextra:              Enables some extra warning flags that are not enabled by -Wall
# Werror:              Make all warnings into errors
# pedantic:            Issue all the warnings demanded by strict ISO C and ISO C++
# Wunused-variable:    Warn whenever a local variable or non-constant static variable is unused aside from its
# Wuninitialized:      Warn if an automatic variable is used without first being initialized
# Wshadow:             Warn whenever a local variable or type declaration shadows another variable
# fstack-protector:    Enable stack protection checks
# Wconversion:         Warn for implicit conversions that may alter a value
# Wunused-function:    Warn whenever a static function is declared but not defined or a non-inline static function
# funsigned-char:      Treat character data type as unsigned instead of signed
# fdata-sections:      Instructs the compiler to place each global or static variable in a separate data section
# ffunction-sections: Instructs the compiler to place each function in a separate code section
set(COMMON_FLAGS
    -g
    -O2
    -Wall
#    -Wextra
#    -Werror
#    -pedantic
    -Wunused-variable
    -Wuninitialized
#    -Wshadow
    -fstack-protector
#    -Wconversion
    -Wunused-function
    -funsigned-char
    -fdata-sections
    -ffunction-sections
)
```

Target Toolchain Files – An Example

```
# C++ flags
#
# std:                  Let's use C++17
# fno-rtti:             Disable Run-Time Type Information. Decreases binary size and improves performance
# fno-exceptions:       For microcontrollers, we disable exceptions due to run-time overhead and dynamic allocation
# lstdc++:              C++ should link our application
set(CPP_ONLY_FLAGS
    -std=c++17
    -fno-rtti
    -fno-exceptions
    -lstdc++
)

string(REPLACE ";" " " CPP_ONLY_FLAGS "${CPP_ONLY_FLAGS}")

# C flags
#
# std:                  Let's use C11
set(C_ONLY_FLAGS
    -std=c11
)

string(REPLACE ";" " " C_ONLY_FLAGS "${C_ONLY_FLAGS}")
```

Target Toolchain Files – An Example

```
# Linker options
#
# -T:                   Linker script
# -specs=nosys.specs:   Linker options for using the Nano Standard C Library
# -Wl,-Map:             Generate a map file
# -Wl,--cref:           Add cross reference to the map file
# -Wl,--gc-sections:    Remove unused sections from the final binary
set(TOOLCHAIN_LINKER_OPTIONS
    -T${LDSCRIPT}
    -specs=nosys.specs
    -Wl,-Map=${EXE_DIR}/${TARGET}.map,--cref
    -Wl,--gc-sections
)

# Libraries to link
#
# c:                    The C library
# m:                    The math library
# nosys:                The Nano Standard C Library
set(TOOLCHAIN_LIBRARIES
    c
    m
    nosys
)
```

Audience POLL Question

Why do you use a target toolchain file when developing embedded software?

a) To define the specific cross-compiler and linker required for the target hardware
b) To configure hardware-specific optimization flags for better performance
c) To manage dependencies and libraries that are specific to the target environment
d) To ensure compatibility with the target operating system or RTOS

Next Steps

05

Embedded Build System

Transform your build system with the free Beningo Embedded Build System example:

- Docker container build system
- Makefile-based
- CMake with Ninja Example
- Compilation scripts
- Integrated tools like cpputest

https://mailchi.mp/beningo/beningo-devops

Additional Resources

Please consider the resources below:

- Jacob's Blogs
- Jacob's CEC courses
- Embedded Software Academy

- Embedded Bytes Newsletter
  - http://bit.ly/1BAHYXm

Consulting | Coaching | Training

in

www.beningo.com

EMBEDDED SOFTWARE ACADEMY BY BENINGO

Next Steps

- ✓ Introduction to Embedded Build Systems
- ✓ CMake Fundamentals
- ✓ CMake for Embedded Systems
- Designing your Build System
- Adopting Modern Practices

Thank You

Sponsored by