CMake and Toolchains: Introduction

ESE PROG-5

HAN_

How to compile a file..

To explain how a normal toolchain works, we start with a normal source file: hello_world.c.
We want to compile and link this to an executable we can run

natively on our pc:
Linux/Mac:
Using the GCC compiler:

```
$ GCC hello_world.c -o hello_world
$ ./hello_world
```

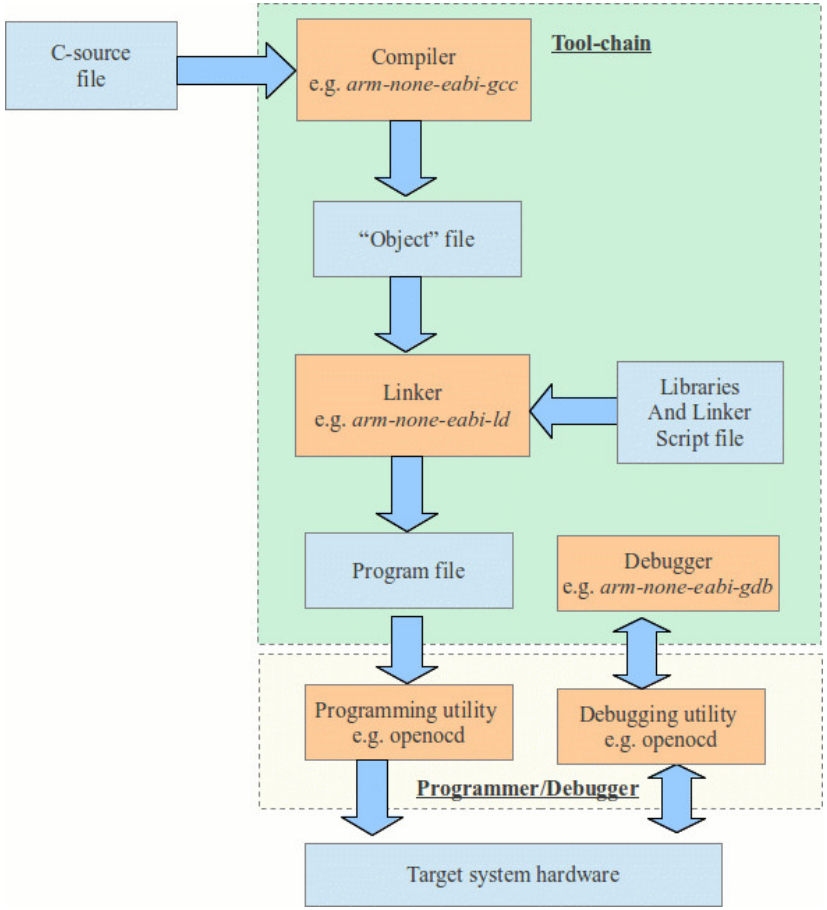
Windows:
Using the MSVC compiler:

```
> cl hello_world.c
> hello_world
```

Toolchain process:

The toolchain will execute the following steps:

1. Compile the source file to a object file (machine code).
2. Link the file with external compiled library files and the linker script (definition of the memory spaces).
3. When developing for embedded systems, utilities like Openocd or the Segger suite of tools might be used for debugging and programming.

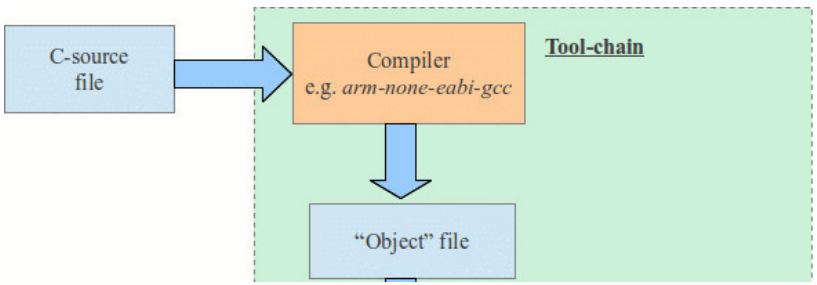


Compiling to object files

The compiler does the following when compiling files in to object files:
1. **Preprocessing:**
- The compiler executes any directives specified in the source code (lines beginning with #, such as #include and #define).
- It includes header files, expands macros, and performs other tasks to prepare the source code for compilation.

2. **Compilation:**
- The compiler translates the preprocessed C code into assembly code.
- It checks the syntax of the C code, ensuring it adheres to language standards, and reports any errors it encounters.

3. **Assembly:**
- The assembler converts the assembly code into machine code.
- The output is stored in an object file with a .o or .obj extension.



Demo: Looking at the preprocessor, compiler and assembler output

GCC can be instructed to print the preprocessor output to the terminal using the -E parameter. To analyze the output we want to print it to a file using the ">" character.

```
% gcc src/main.c lib/i2c.c lib/spi.c lib/uart.c
-I lib/ -E > preprocessor_output
```

Answer

3 Times! We have 3 source files utilizing the printf function. Each source file will be compiled separately and linked together.
Meaning 3 copies of the printf are needed to make it compile.

Let's now take a look at the object files... Again gcc can be instructed to only compile and not link the object files in a binary... What does the binary hide?

We can do this using the gcc command with -C option (this forces compiler not to link):

```
% gcc src/main.c lib/i2c.c lib/spi.c lib/uart.c -c
```

And then observing it with objdump..

Question

Without optimization flags and special compiler flags, how does the compiler handle constant strings inside the printf statement?

Remark: Memory sections

When a program is compiled and linked, the resulting executable contains various pieces of information that are used both at load time and runtime, such as the program's code, data, and additional metadata. Object files have a specific memory layout that organizes these different types of information into various sections. Understanding this memory layout is essential for debugging and optimizing compiled programs.

Basic sections

Text Segment (`.text`): Contains the executable code (machine instructions) of the program.
Typically marked as read-only to prevent the program from accidentally modifying its own instructions.

Data Segment: Further divided into the Initialized (`.data`) and Uninitialized (`.bss`) Data segments.

`.data`: Contains global and static variables that are initialized in the code. It holds the actual data values.
`.bss`: Contains global and static variables that are not explicitly initialized in the code. This section is typically zero-initialized by the operating system at runtime.

Read-Only Data Segment (`.rodata`): Contains constants and string literals which should not be modified during the execution.
Marked as read-only.

Heap: Memory segment used for dynamic memory allocation.
Grows at runtime as more memory is dynamically allocated (e.g., with `malloc` in C).

Stack: Contains function call frames, local variables, and control information for function calls.
Used for managing function calls and returns.

Additional sections

Symbol Table (symtab): Contains information about functions and variables in the code. Used by the linker and debugger.

Debug Information: Contains source file names, line numbers, and other debug information. Used by debuggers to map machine instructions back to source code lines.

Relocation Table: Contains information used by the linker to adjust the program's memory addresses.

The linker

How to compile multiple files?

Imagine the following use-case:
We want to compile a library with one main source file and a library containing multiple source files and header files.
The target is a RPI pico W microcontroller, the compiler used is

the GCC compiler.

```
Project folder
├── lib
│   ├── i2c.c
│   ├── i2c.h
│   ├── spi.c
│   ├── spi.h
│   ├── uart.c
│   └── uart.h
└── src
    └── main.c
```

To compile all these files we would need to enter the following command:

```
$ gcc-arm-none-eabi main.c lib/i2c.c lib/spi.c
lib/uart.c -I lib/ -mcpu=cortex-m0plus
-mthumb -O main.hex
```

As you might have noticed the length of the command has increased noticeably.
This of course doesn't work out well when having more files or/and

having to set target specific options like error-checking and warnings.
What if there would be a solution to this...

Make

- Make is a build automation tool used to manage and automate the build process of a project.
- It uses a Makefile to specify how a program should be compiled and linked.
- Make is very popular in the (embedded) Linux community as it is provided by default in the package repository's, and used when consistency is required.

How does Make work?

Makefiles are nothing but a recipe..
It always contains:
- **Target**: The output file to be built.
- **Dependencies**: Files that the target depends on.
- **Recipe**: A series of commands to produce the target from the dependencies.
Example of a rule:

```
target: dependencies
recipe
```

A very important thing Make will do is check the time stamps of the target file and dependencies. If no file was changed then make will exit, else it will continue with the next step: **run the recipe commands**.
Make executes the recipe commands to create the required target.

Makefile example for use-case defined in previous slide

```
# Compiler and Linker
CC = arm-none-eabi-gcc
LD = arm-none-eabi-ld
# Flags
CFLAGS = -mcpu=cortex-m0plus -mthumb

# Include paths, assuming the Pico SDK is installed and PICO_SDK_PATH is set
INCLUDE = -I$(PICO_SDK_PATH)/include

# Source files
LIB_SOURCES = lib/i2c.c lib/spi.c lib/uart.c
MAIN_SOURCE = src/main.c

# Object files
OBJS = $(LIB_SOURCES:.c=.o) $(MAIN_SOURCE:.c=.o)

# Output executable
OUTPUT = main.elf

# Default target
all: $(OUTPUT)

# Rule to compile source files to object files
%.o: %.c
	$(CC) $(CFLAGS) $(INCLUDE) -c $< -o $@

# Rule to link object files and create executable
$(OUTPUT): $(OBJS)
	$(LD) $(LDFLAGS) $(OBJS) -o $(OUTPUT)

# Clean target
clean:
	rm -f $(OBJS) $(OUTPUT)
```

When to use Make and what are the limitations

When to use:
- Simple to moderately sized projects
- When you need full control over the build-process
- Projects primarily targeting or using the Unix or GNU+Linux operating system

Limitations:
- Makefiles are platform and compiler specific.

Meaning you have to write different Makefiles for different platforms, or include complex conditional logic in a single Makefile.

Microsoft Visual Studio solution files

A solution file for c/c++ projects most of the time have a .vcxproj file extension.
A solution files is composed of a XML file describing the build targets, compiler settings and build environment specifics.
Solution files are only used under Windows and can't be used

natively under any other operating system.
Due to the XML format, visual studio solution files are pretty easy

to understand, however they don't give you the absolute control a Makefile would give you.

Solution file example

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build" ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/

  <!-- Global Properties -->
  <PropertyGroup Label="Globals">
    <ProjectGuid>{CE8885D6-7C94-40B7-8F0A-0C5ED9CFA1F5}</ProjectGuid>
    <Keyword>Win32Proj</Keyword>
    <!-- Other Global Properties -->
  </PropertyGroup>

  <!-- Configuration Properties -->
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'" Label="Config
    <ConfigurationType>Application</ConfigurationType>
    <!-- Other Configuration Properties -->
  </PropertyGroup>

  <!-- Import Default Settings -->
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />

  <!-- Other Settings and Configurations -->

  <!-- List of Files to Compile -->
  <ItemGroup>
    <ClCompile Include="main.cpp" />
    <!-- Other Source Files -->
  </ItemGroup>

  <!-- Import Targets -->
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />
</Project>
```

When to solution files and what are the limitations

When to use:

- **When everyone is working with visual studio.**

Some advanced features like IntelliSense, GUI-based configuration and some of the debug tools are only available when using solution files in visual studio.

- **When developing for Windows.**

For software targeting Windows platforms, .vcxproj files provide easier setup and integration with Windows-specific libraries and frameworks.

- **When working with Mixed Languages.**

Visual Studio supports multiple languages (C++, C#, F#, etc.) in a single solution, making it easier to work with mixed-language projects.

Limitations:

- **Very Windows bound..**

Solution files are closely tied to the Microsoft ecosystem.
They are not as platform-agnostic as some other build systems.

- **Complexity**

Although XML is very human readable, it will become very unreadable quickly when having to build big projects (lots of files) with a lot of project-specific settings.

- **Less Fine-grained Control**

.vcxproj files may not offer as much fine-grained control over the build process as some other systems, like make.

Ninja build files

What if I told you there is another tool, yes another one...
Introducing Ninja...
Ninja is a small build system that is designed to be fast. It focuses on improving the speed of incremental builds, where only a subset of the source files are modified. It was originally designed to speed up the builds for Google Chrome and is used in the Android operating system's build system as well.

How Ninja works

- Ninja uses a simple text file (build.ninja) as a build file.
- This file contains rules and dependencies for building the project.
- Unlike other build systems, build.ninja files are not written by hand. Instead, they are typically generated by a higher-level build system like CMake or Meson.

Build.ninja example

```
# The Ninja build file for compiling the hello project.

# Variables.
cc = gcc
cflags = -Wall -O2
lflags = -o

# Rule to compile C files.
rule cc
  command = $cc $cflags -c $in -o $out

# Rule to link object files into an executable.
rule link
  command = $cc $lflags $out $in

# Build edges.
build main.o: cc main.c
build hello.o: cc hello.c

# Link the object files into the final executable.
build hello: link main.o hello.o
```

When to use Ninja and what are the limitations

When to use:

- **Large codebases**

For extremely large codebases, Ninja can significantly reduce the build time, especially for incremental builds, as it focuses on improving the speed of such builds.

- **Cross-Platform Development**

Ninja is platform-agnostic, making it a good choice for projects that need to be built on multiple operating systems.

- **Complex dependencies**

Ninja effectively manages complex dependencies in a project, ensuring that only the necessary parts of the project are rebuilt, which helps in saving build time.

Limitations

- **Limited Functionality**

Ninja has fewer features compared to other build systems like CMake or Make. It's generally used as a backend to these systems and doesn't handle configuration or project setup.

- **Less Ideal for Small Projects**

For smaller projects, the overhead of setting up Ninja (or another system to generate Ninja build files) might not be justified, and simpler build tools might be more appropriate.

Recap:

Each build system has its own strengths and weaknesses.
When we recap on previous slides we can conclude the following:
Use make when:

You want cross-platform compatibility and fine-grained control over the build process, suitable for smaller to medium projects.

Use .vcxproj when:
Working primarily on Windows, using the Visual Studio IDE, and dealing with multi-language projects.

Use Ninja when:
Working with large codebases where fast build times are crucial, especially when using build systems like CMake or Meson that can generate Ninja build files.

Ultimately, the best choice depends on your specific project requirements, including the size and complexity of your project, your target platform, and your preferred development tools and environment.

How do I decide?

Good question!
The answer is... *drumroll*
It depends!

If requirements are not too strict on the toolchain, there are tools that make this decision *cough* *cough* easier.

Easier in the sense that the tool makes the decision for you.
Take eclipse CDT or CMake

CMake: Introduction

Ah yes finally. Let's skip over eclipse anyway, as it is not a worthy contender anyway...
But what is CMake?

CMake is a tool which **generates build files**. It mostly used in conjunction with native build environments such as Make, Apple's Xcode, and Microsoft Visual Studio.

CMake uses it's own scripting language (CMakeLists.txt files) to define build settings, targets, and dependencies.

Although CMake is very well documented, it still might require some getting used to (just like any language :p).

CMake: example CMakeLists.txt

CMake has both separate build scripts for libraries and the main project source files.

Project CMakeLists.txt

```
# Set the minimum required version of CMake
cmake_minimum_required(VERSION 3.10)

# Set the project name
project(Hello_world)

# Add the executable
add_executable(Hello_world src/main.c)

# Link the executable with the library
target_link_libraries(Hello_world PeripheralDriver)
```

Library CMakeLists.txt

```
# Add library source files
add_library(PeripheralDriver STATIC lib/i2c.c
                                    lib/spi.c
                                    lib/uart.c)

# Add current folder "lib/" as include directory
target_include_directories(PeripheralDriver "." )
```

CMake: Syntax deep dive

- **Comments:**
  Start with a # symbol.

```
# This is a comment
```

- **Variables:**
  Variables are case-sensitive and can be set with the set() command.

```
set(MY_VARIABLE "Hello, World!")
```

- **Commands:**
  Commands are case-insensitive and called using parentheses ().

```
# Outputs a message during the cmake generation step
message("This is a message")
```

- **Control Structures:**
  CMake supports control structures such as if, while, and foreach.

```
if(MY_VARIABLE STREQUAL "Hello, World!")
    message("Variable is Hello, World!")
endif()
```

- **Project Information:**
  Set the project name and version. Can only be set once in a project, is mostly set right after setting the minimum cmake version.

```
project(MyProject VERSION 1.0)
```

- **Include directories:**
  Sets include directory of the executable target.

```
include_directories(${CMAKE_SOURCE_DIR}/include)
```

- **Add Executable:**
  Define an executable target and its source files.
  ```
  add_executable(MyExecutable main.cpp util.cpp)
  ```
- **Add Library:**
  Define a library target and its source files.
  ```
  add_library(MyLibrary STATIC lib1.cpp lib2.cpp)
  ```
- **Target Link Libraries:**
  Link libraries to targets.
  ```
  target_link_libraries(MyExecutable MyLibrary)
  ```
- **Custom Commands and Targets:**
  Define custom build commands and targets.
  ```
  add_custom_command(...)
  add_custom_target(...)
  ```

CMake: Advanced commands and features

CMake is a scripting language, that means it can also download dependencies like external libraries from GitHub.

```
# Include the ExternalProject module
include(ExternalProject)

# Set the minimum required version of CMake
cmake_minimum_required(VERSION 3.11) # ExternalProject_Add() requires at least CMake 3.11

# Set the project name
project(MyProject)

# Add the external project
ExternalProject_Add(
    name_of_the_external_project                # Name for the external project
    PREFIX ${CMAKE_BINARY_DIR}/_deps            # Download location
    GIT_REPOSITORY https://github.com/user/repo.git  # URL of the GitHub repo
    GIT_TAG main                                # Git branch, commit or tag to checkout
    CMAKE_ARGS -DCMAKE_INSTALL_PREFIX=<INSTALL_DIR>  # Installation directory
)

# You can then link against or add dependencies as needed
add_dependencies(your_target name_of_the_external_project)
```