Auomated Embedded Project Setup

by Søren Kjædegaard Haug
exam number: 486019

Final Project
A Project Submitted for the Degree of
B.Eng. Electronics

In cooperation with:
LINAK A/S

SDU Supervisor:	LINAK Supervisor:
Søren Top	Anders Kristensen

Sønderborg
January 2nd 2023

02/01-2024
Søren K. Haug Date

Glossary

ABI	Application Binary Interface.
API	Application Programming Interface.
CMSIS	Common Microcontroller Software Interface Standard.
CRC	Cyclic Redundancy Check.
DLSW	DESKLINE Software.
ELF	Executable and Linkable Format.
IDE	Integrated Development Environment.
LTO	Link Time Optimization.
MCU	Microcontroller unit.
OFTC	The Open and Free Technology Community.
stderr	Standard Error.
stdout	Standard Output.
VFP	Vector Floating Point.

Contents

List of Code Blocks

List of Figures

List of Tables

Chapter 1

Introduction

LINAK's DESKLINE segment is experiencing an increasing challenge in the development process for its software. With a growing software platform, the time it takes developers to set up and build projects is increasing. Manual project setup can lead to human errors, which can further increase the amount of resources spent during this process. DLSW (*DESKLINE Software*) is currently limited to using the IDE (*Integrated Development Environment*) μVision Keil to set up, build, and debug their software. The use of μVision Keil also has a considerable annual cost associated with it.

One way to improve the work process of individual developers and reduce the cost associated with building a software project is by implementing a build automation tool. Automating the build process can help reduce the time an individual developer uses to set up projects, almost eliminate human errors, and improve overall development efficiency. The goal of this project is to investigate select available build automation systems and present a demonstration software showing a solution to achieve build automation.

1.1 About LINAK

LINAK was established by the current CEO and owner Bent Jensen's grandfather in 1907. After Bent Jensen took over the company from his father in 1976, he revolutionized it by inventing the electric linear actuator in 1979. With the introduction of this technology, LINAK quickly established itself as a leader in the industry.

Today, LINAK is still an industry leader with a commitment to innovation and a commitment to providing the highest quality products and services to customers around the world. In 2019 LINAK counted more than 2400 employees spread over 35 countries. The headquarters is located in Guderup in southern Denmark and covers an area of 55,000m^2. 1250 employees work at the Guderup headquarters in manufacturing, group management, R&D, logistics, and administrative functions. LINAK has established factories outside Denmark in the USA, China, Slovakia, and Thailand. [1]

LINAK consists of 4 business segments.

- DESKLINE specialises in ergonomic systems for office desks, workstations, kitchens, and shop and conference room interiors
- HOMELINE excels within comfort furniture
- TECHLINE handles durable actuator solutions for heavy-duty applications operating in the harshest of environments
- MEDLINE & CARELINE focuses solely on developing intelligent systems for healthcare applications

LINAK was born from an innovative idea and is built on the wish to convert new ideas into innovative products. It has a large team of experienced and highly skilled engineers to maintain the best possible foundation to create meaningful solutions and value-adding products for its customers. [2]

1.2 Objectives

To achieve the goal, several objectives have been identified. The general objectives can be found in the problem formulation in Appendix A. The detailed objectives are listed below, prioritized into 4 categories; *Must Have*, *Should Have*, *Could Have*, and *Won't Have*.

1.3 Detailed objectives

1. **Must Have**
   (a) Investigation of build automation and Makefiles.
   (b) Analysis of two different build automation solutions.
      - CMake and Meson.
   (c) Implementation of a demonstration software:
      - Must generate a project *.hex file using Arm Compiler 6 toolchain.
   (d) Integration with an SVN server to fetch desired SW.
2. **Should Have**
   (a) Include project setup using DLSW's CLEAR SW platform.
   (b) Ability to easily switch between different compilers.
      - Arm Compiler 5 (armcc), Arm Compiler 6 (amrclang), gcc
   (c) Be independent of the use of a specific IDE.

3. **Could Have**
   (a) Options to set different compiler settings (optimization etc.).
   (b) Investigation and analysis of additional solution(s).
   (c) Have the prototype software running on a Jenkins server.
   (d) A method to debug a *.hex file, built using the chosen solution, on a hardware target.

4. **Won't Have**
   (a) Continuous delivery/deployment analysis.
   (b) Production SW solution for DLSW.

1.4 Work Methodology

The approach to the project will be similar to that of the V model. To start the project, all relevant theories of automation software will be investigated. Based on this investigation, two systems will be subject to further investigation. This will lead to an analysis of the two systems, comparing them to each other, and presented to DLSW for the strategic choice of which one to go in to development. Then a demonstration project will be created that presents the chosen solution, followed by verification and evaluation.

1.4.1 Risk Assessment

Before the project began, all relevant risks were identified. Each risk was rated according to probability and impact on the project. For each of the risks, a mitigation strategy was identified. Risk tables can be found in the Appendix A.1

1.4.2 Initial Time Schedule

The initial time schedule shows the main activities of the project, any subcategories of these, and allocates time to each phase. It is initially divided into 3 main phases, **Research and Investigation**, **Analysis** and **Demonstration SW**. These reflect the main activities and roughly the different parts of the project report.

There are 3 milestones in the project, marked with ♦ in the schedule.

1. 11th of October: Criteria for automation system agreed upon by all stakeholders
2. 21st of November: Choice of automation system for demonstration software development
3. 11th of December: Presentation of demonstration software.

The time schedule can be found in Appendix A.2

Chapter 2

Background & Analysis

This chapter provides background information on some of the build system tools and generators used to build and describe software projects. It includes a list of requirements for a build system to be considered feasible and compares the two build system generators, CMake and Meson. At the end of the chapter, an evaluation matrix is presented, summarizing the two build system generators side by side.

2.1 Build Systems

A build system is a tool that provides a consistent and reusable framework to define and automate the software development, quality control, and delivery process. The system defines how software and other components are assembled, the dependencies that must be satisfied, and any third-party components that are used in the build process.

One of the main features of a build system is having a common method to define the build process and invoke it. The build system should work the same across different projects and is expected to behave in the same way on each project.

Terminology for build systems and their meaning within a build system context:

- A **build machine** is the computer that executes the build process
- A **host machine** is the computer on which the software will execute. This may or may not be the same as the *build machine*.
- A **target machine** is the machine on which the compiled output will run, which is only meaningful if the program on the *host machine* produces machine-specific output (e.g. cross-compilation). For most cross-compiled programs, the *target machine* and the *host machine* are the same thing.

Within compilation there are also different types to account for:

- **Native compilation**: The software is compiled so it can run on a host machine that is the same architecture as the build machine
- **Cross-compiling**: Involves compiling software that will run on a host machine with a different architecture than the build machine. for example, compiling for ARM Cortex-M0+ using an Intel x86_64 machine.

A **toolchain** is a set of tools for producing software, such as compiler, linker, archiver, etc. the term **toolchain** refers to the sum of these parts.

A build system is a tool used in software development to manage parts of the process. Software development is a complicated process which includes tasks such as source code control, code generation, automated source code checks, documentation, compilation, linking, unit testing, integration testing, packaging, creating binary releases, source code releases, deployment, and reports.

This can be simplified into four main phases of development.

1. Developers write source code.
2. The source artifacts are transformed to end products.
3. The end products are tested.
4. The end products are deployed or distributed.

A good automated build system can manage steps two through four.[24]

Apart from providing technical advantages and capabilities for the development progress, the use of a build automation system also has several human factor benefits.

A build system will provide a standardized and automated approach to building the target software. Without this automation, the manual process is error-prone. Some common errors can include developers forgetting to run a "clean" build and/or switching to the release branch. This will result in a build that does not reflect the current state of the release source tree. The manual process requires the developer to carefully build, test, and package the software before handing it over to QA. If a single step is done incorrectly, the build will be sent back to the developer and QA will have wasted time testing the wrong software.

When the build, quality enforcement, and packaging processes are automated, there is no chance that a developer misses a step or builds the wrong branch.

2.2 Make

Source code by itself does not make an application, and the way they are put together and packaged for distribution matters. Make is a tool for automating these processes. Make automates

the building of executable programs and libraries from source code and their dependencies. It reads files known as **Makefiles** which contain instructions on how to derive the target program. When developing in C or C++, an important part of building an application is the collection of compilation and linkage commands needed to get from the source code to working binaries. Entering these commands can be a lot of tedious work. Make was designed to help C programmers manage these commands.

Make can be used to manage the build processes of complex projects and helps to decide which parts of a large program need to be recompiled. It is particularly popular in UNIX and UNIX-like environments but is also available for Windows and other operating systems.

Make was first created by Stuart Feldman in April 1976 at Bell Labs. Feldman was inspired to write Make from an experience of a coworker who was debugging a program of his, where the executable was accidentally not being updated with changes to the source tree. [23]

2.2.1 Makefiles

A Makefile is a plain text file that describes how to build a program or project. When invoking Make commands, it searches the current directory for a Makefile and invokes the specified targets from the file. Makefiles allow the developer to declare dependencies between files in a project. They can specify that a particular source file depends on one or more header files, which object files should be linked to libraries, and specify the output executable. Through timestamps of the source and header files, Make determines which parts of a project need to be recompiled or not.

The basic syntax of a makefile is as follows:

```
target: prerequisites
    recipe
```

The `target` is the name of the output file that to be created. The `prerequisites` are the files that the targets depends on. The `recipe` is the shell command used to create the target. If necessary, several prerequisites are listed after each other separated by a space.

For example the following would create an executable file called `bin` from the source code in `x.c`

```
bin: x.c
    gcc x.c -o bin
```

First the timestamp of the dependency is checked, and if it has changed from any previous builds, the `make bin` command will execute the recipe in the Makefile to create the target.

Makefiles are typically written in a simple text format, which makes them humanly readable and capable of being developed by a human user. They can be used to build projects that use multiple different languages and can also be used to automate other tasks, such as unit tests.

2.3 Ninja

Ninja is a build system similar to Make, although there are noticeable differences between the two. It differs from Make and other high-level language build systems in that it is designed to have its input files generated by a higher-level build system, like CMake or Meson. Ninja's build files are human-readable but not especially convenient to write by hand.

Ninja was created by developer Evan Martin who, at the time, worked on Google Chrome. Evan discovered that when they used `Scons` and later `Make` to build Chrome, it could take as long as 40 seconds from starting the process, before the source code started to build. It could take up to eight minutes to complete linking after altering just one file. With a want to bring down building time, he began developing Ninja to be similar to Make, but without hardly any features. Combined with fast linking tools and a fast computer, an incremental build of Chrome was clocked at 6 seconds when altering just one file. [13]

The design goals of Ninja are

- Very fast incremental builds.
- Very little policy on how code is built. This is left to the higher-level build systems.
- Get dependencies correct (fixing some implicit dependency issues found when using Make).
- When convenience and speed are in conflict, prefer speed.

When Ninja was created an explicit non-goal was to not have convenient syntax for writing the build files by hand. Ninja files should be generated by another program or build system generator.

Ninja builds are run in parallel by default with the number of jobs depending on the system CPU. The number of jobs can be set when invoking Ninja with the `-jN` flag, where `N` is the number of concurrent jobs.

2.4 CMake

CMake was created in 2001, is an open source system, and is currently the most widely used build system for C and C++. CMake is a suite of tools that covers everything from setting up a build right through to producing packages ready for distribution. CMake is a build system generator, which means that CMake itself does not execute the software building process. Instead, CMake translates a high-level build description into a format accepted by other tools. It provides generators for many popular build tools, including Make, Ninja, Xcode, Visual Studio, etc.

CMake has gone through several larger updates throughout its current life cycle, with the largest jump being from CMake v2 to CMake v3. CMake 3.0 made changes to the syntax to

make it more concise and easier to read, and it is considered the earliest version of modern CMake. CMake v3 was released in June 2014 [14]. Developers are encouraged to always remain relatively close to the most recent stable CMake release [26].

Where Make uses Makefiles, a CMake project uses at least one CMakeLists.txt file to define the build. The build can be defined in a single file, or several can be spread out in the source tree.

CMake build stages:

1. **CMakeLists.txt**: Define the project structure, settings, and dependencies.
2. **Configuration**: Configure build system files (e.g. Makefiles, Ninja) based on the CMakeLists.txt file.
3. **Generation**: Generates build system files (e.g. Makefiles, Ninja).
4. **Build**: Invokes the back-end build system to build the project.

2.4.1 CMakeLists.txt

The CMakeLists.txt file describes the source files and the targets of a project, such as executables, libraries, tests, etc., and how they should be compiled and linked. It also specifies the options and properties of the project, such as the minimum required version of CMake, the languages used in the project, the language standard, etc. The CMakeLists.txt file is usually located in the root directory of the project source code. additional files can also be placed in subdirectories to describe different modules or components of the project.

The syntax used in CMake is based on its own CMake language, which consists of commands and variables. Commands are used to specify actions and options for building the project. The commands in CMake closely resemble functions known from languages like C and C++. Variables are used to store and manipulate values that can be referenced throughout the project, depending on the scope of the variable.

```
cmake_minimum_required(VERSION 3.15)
# Set Project Details
project(myProject
    VERSION 1.00
    DESCRIPTION "My Project"
    LANGUAGES C ASM
)
# Add executable with main.c
add_executable(myApp main.c)
# Add source.c to SOURCE_FILE variable and target the executable
set(SOURCE_FILES source.c)
target_sources(myApp PRIVATE ${SOURCE_FILES})
# Link executable with myLib
target_link_libraries(myApp PRIVATE myLib)
```

Code Block 2.1: CMakeLists.txt Example

2.5 Meson

The creation of Meson started at the end of 2012 by the Finnish developer Jussi Pakkanen. He was frustrated with the existing build systems due to their "foolish configuration syntax and unexpected behavior" [25]. The system had its initial release on 2 March 2013, and has since gone through a series of updates, with the latest release, version 1.2.3, released on 20 October 2023. The release notes for the versions of the can be found on the Meson website [20].

Meson is a build system generator and generates build files for build systems like Ninja, Visual Studio, and Xcode. It is created with multi-platform support for Linux, MacOS, Windows, etc. and supports languages like C, C++, Rust, etc. It is designed to be very readable and user-friendly, and the main design principle is that it is non-Turing-complete. It supports cross-compilation for many operating systems, bare metal included. It is optimized for extremely fast full and incremental builds, while promising correctness in the output. [20].

A Meson project uses at least one `meson.build` file to define the build. The build can either be described in a single file or spread throughout the source tree.

To build a project with Meson, a build output folder is specified when invoking Meson. Meson will create the target output folder if it does not already exist and will then use the build files, any configuration settings supplied by the user, and the chosen back-end build system to produce a build system tuned for the platform and configuration. The generated files are placed within the build output folder. The build options are stored in Meson files in the build output folder, so they are remembered between build invocations.

Every Meson project must have a project definition; This is usually declared in the root `meson.build` file, and declares the project name, programming languages, license, project version, required meson version, and more. default build options can also be supplied in the project declaration. It is possible to override these options if desired.

```
# Set Project Details
project('Project Name',
    ['c'],
    default_options : [
        'c_std=c99',
        'warning_level=3'
    ],
    license: 'MIT'
    meson_version: '>=1.2.0',
    version: '1.0'
)
```

Code Block 2.2: Meson.build Example

2.6 Analysis

This section lists the requirements for the build system that must be met before a build system can be considered for further research and development. It compares the two build systems, CMake and Meson, on each of the identified requirements.

To decide which features and requirements should be set up for the build automation system, a meeting was held on October 10 2023. The meeting was scheduled with the DLSW departments manager, architects, and select developers. A list of possible requirements was sent to participants prior to the meeting. The list was discussed and further specified. The summary of the meeting can be found in the Appendix section C.1.

This meeting led to a list of required build system capabilities and non-functional requirements.

Requirement Glossary

- **Binary code:** Typically ELF (*Executable and Linkable format*) or hex format, when cross-compiling.
- **Statistics:** Static, coverage, complexity analysis, sanitizer, documentation, etc.
- **Out-of-source build:** Build files and output are placed in a separate directory.

1. Capabilities

1.1 **Source to binary**

1.1.1 The system must be able to convert the source code to binary code.

1.1.2 The system must provide an output in hex format.

1.1.3 The system must be able to generate statistics for the built software and its structure.

1.2 The system must make it possible to define different build types (debug, release, etc.).

1.3 The system must be able to create out-of-source builds.

1.4 The system must have the ability to build incrementally.

1.5 The system must be able to invoke cross-compiling.

1.6 **Build and host machine**

1.6.1 The system must be capable of running on a Windows x86_64 machine.

1.6.2 The system must be able to cross-compiling for the following host machines.

1.6.2.1 Nordic nRF52810

1.6.2.2 Nordic nRF52832

1.6.2.3 ST STM32F03x

1.6.2.4 ST STM32G03

1.7 The system must be able to fetch content from an SVN server and from GIT.

1.8 **Dependency management**

1.8.1 The system must have the capability to build different project versions based on a specified revision number.

1.8.2 The system must enable the selection of module versions based on a specified revision number.

1.9 The system must be allowed for commercial use, preferably at no cost.

2. Non-functional Requirements

2.1 **Repeatability**

2.1.1 The system must ensure that the same building steps are executed consistently each time the build is run, regardless of the build machine or invoker.

2.2 **Reproducibility**

2.2.1 The system must be capable of generating an identical hex output file when the same configuration is used.

2.3 **Correctness**

2.3.1 The system must produce an updated output, when changes are made within the source tree.

2.4 **Standard**

2.4.1 The system should have widespread industry usage.

2.4.2 The system should have comprehensive documentation.

2.4.3 The system should have strong community support.

2.5 **Debug of build system**

2.5.1 The system should provide warning and error messages that are helpful for locating errors in the system.

2.6 **Ease of use**

2.6.1 The system should not require an extensive setup process when set up with a project template.

2.6.2 The system should be able to invoke a specific build using a single command.

2.6.3 The system should provide an easy-to-learn interface for invoking builds.

2.6.4 The system should have an easy-to-understand syntax for developers not maintaining it on a day-to-day basis.

2.7 **Performance**

2.7.1 The system must not exhibit noticeable performance lag compared to the currently used Keil system.

2.8 **Future-proof**

2.8.1 The system must be developed and maintained by a team with a consistent and reliable history.

2.8.2 The system should be actively maintained by a sufficient number of developers.

2.8.3 The system should support the ability to switch compiler versions.

2.8.4 The system should allow users to suggest new features, report bugs, and provide feedback.

2.8.5 The system must be used by a defined user base in the industry.

2.9 **IDE compatibility**

2.9.1 The system should be supported in IDE's with syntax highlighting, code completion and hover information.

2.9.1.1 Support in VS Code

2.9.1.2 Support in Eclipse

2.9.1.3 Support in Keil

2.6.1 Capability Requirement Comparison

req#1.1.1

"The system must be able to convert the source code to binary code"

Common

Neither CMake nor Meson handles the compilation or generation of binary code directly. They are built system generators who generate platform-specific build files (Makefiles, Ninja build files, etc.).

CMake

CMake handles this through a set of `CMakeLists.txt` files in the root of the project and the necessary subdirectories, depending on the size and scope of the project. In the CMakeLists the source files, libraries, compile options, possible output targets, etc. are defined. Through the specifications in the CMakelists, CMake generates platform-specific build files that handle the conversion of the source code to binary code.

Meson

Meson handles this similarly to CMake but relies on the `meson.build` files in the root of the project and the necessary subdirectories. Like in the CMakelists, the various source files, libraries, options, etc, are defined in the meson.build files.

req#1.1.2

"The system must provide an output in hex format"

Common

Neither CMake nor Meson has built-in functionality to convert output files to hex files, but relies on the given toolchain to perform this operation. If the toolchain used does not include a means of converting the output executable or ELF file, the GNU Binutil tool `objcopy` can be used.

To convert an ELF file to a hex file using `objcopy`, the command must be used:

```
1 objcopy -O ihex <input> <output>.hex
```

The `-O ihex` specifies the output format as `Intel HEX`.

The Clang toolchain uses `Fromelf` to do the conversion

```
1 Fromelf --i32combined --output=<output>.hex <input>.elf
```

CMake

A variable in the toolchain can be set to save the toolchain conversion executable, which can be referenced later to perform the conversion.

```
set(OBJCOPY     arm-none-eabi-objcopy)
```

To trigger the conversion from an output file to hex, a user-defined function can be made, and within that a custom command to perform the conversion can be invoked. The `COMMAND` option specifies the command line(s) to execute at build time.

```
function(convert_to_hex TARGET
    add_custom_command(
        TARGET ${TARGET} POST_BUILD
        COMMAND ${CMAKE_OBJCOPY} -O ihex ${TARGET} ${TARGET}.hex
        BYPRODUCTS ${TARGET}.hex
    )
endfunction()
```

Code Block 2.3: CMake custom function to convert TARGET to hex format

Meson

Through the built-in meson object, an external property can be found in the specified cross- or native file. (See req#1.6.2 meson description).

```
objcopy = find_program(meson.get_external_property('objcopy'))
```

Then a custom target can be created where the input, output, and the command to run are specified. If the target must be built at every run, the `custom_target` variable `build_by_default` must be set to true.

```
sample_app_hex = custom_target('sample_app.hex',
    input: sample_app,
    output: 'sample_app.hex'
    command: [objcopy, '-O', 'ihex', '@INPUT@', '@OUTPUT@'],
)
```

Code Block 2.4: Meson custom function to convert TARGET to hex format

req#1.1.3

"The system must be able to generate statistics for the built software and its structure"

Common

Both CMake and Meson have ways of defining and running different tools, that are not built-in to the system itself. These often include the creation of a separate shell script, and a built-in function is used in one of the two systems to run the script.

CMake

CMake has some built-in support for static analysis tools such as `clang-tidy`, `CPPLint`, `CPPCheck`, `Include What You Use`, etc. Each property or variable is a CMake list of arguments that are used to invoke the tool, including the binary. The tool is then automatically invoked when building targets that have the properties enabled.

An example using built-in `cppcheck` to set the CMake variable `CMAKE_C_CPPCHECK`:

```
find_program(CPPCHECK cppcheck)

if(CPPCHECK)
  set(CMAKE_C_CPPCHECK
    ${CPPCHECK}
    --quiet
    --enable=style
    --force
    -I ${CMAKE_CURRENT_LIST_DIR}/lib
  )
endif()
```

Code Block 2.5: CMake CPPcheck setup

CMake has limited built-in support for code coverage analysis, where it supports `gcovr` if `CTest` is used.

Meson

Meson only supports the Clang static analyzer `scan-build`, which can be called directly when executing the build files. When using the Ninja back-end, it can be invoked with the command:

```
ninja scan-build
```

It has built-in support for two different code coverage analysis tools `gcovr` and `lcov`, which can be enabled with the `b_coverage` variable, when configuring the project.

It has built-in support for a set of different sanitizers, which can also be set up when configuring the project through the `b_sanitize` variable. The value can be either `none`, `address`, `thread`, `undefined`, `memory` or `leak`. The sanitizer will only be used if it is supported by the compiler.

req#1.2

"The system must make it possible to define different build types (debug, release, etc.)"

Common

Both systems have an option to choose between several predefined build variations. Custom variations can be defined in both.

CMake

CMake has a built-in variable called `CMAKE_BUILD_TYPE`, which can be set to either `Debug`, `Release`, `RelWithDebInfo` or `MinSizeRel`. The variable is initialized in the project configuration and the default value is an empty string. If the string variable is empty, the toolchain-specific default is chosen. As an example, `MinSizeRel` includes the following c flags `-Os -DNDEBUG`

Meson

Meson has a built-in core option to define the build type to use. The variable is called `buildtype`, and can contain the values: `plain`, `debug`, `debugoptimized`, `release`, `minsize` and `custom`. Each of these options has its own pair of true/false debug and optimization levels. For example, `debugoptimized` sets the internal variables `debug=true` and `optimization=2`. [16]

req#1.3

"The system must be able to create out-of-source builds"

Common

Both systems are easily able to produce out-of-source builds.

req#1.4

"The system must have the ability to build incrementally"

Common

Both CMake and Meson provide incremental building support by generating build files that include dependencies between targets and source files. Incremental building depends on the chosen underlying build system, but the most common build tools like `Make`, `Ninja`, `Visual Studio`, and `Xcode` supports incremental building.

Meson

From the description of the Meson features:
"optimized for extremely fast full and incremental builds without sacrificing correctness"[20]

req#1.5

"The system must be able to invoke cross-compiling"

CMake

To initiate cross-compilation in CMake, a valid toolchain file must exist and it must be invoked with one of two command line parameters. [9]

```
cmake --toolchain path/to/toolchain_file.cmake path/to/source
```

```
cmake -DCMAKE_TOOLCHAIN_FILE=path/to/toolchain_file.cmake path/to/source
```

The CMake variable `CMAKE_CROSSCOMPILING` is set to true when the variable `CMAKE_SYSTEM_NAME` has been set manually.

However, it is recommended to avoid using the `CMAKE_CROSSCOMPILING` variable, as it can be misleading and set to true when the build and target systems are the same or false when they are different. [26]

Meson

Meson cross-compilation initialization is very similar to CMake, but the way it handles cross-compilation is slightly different. to initiate cross-compilation, a valid cross-file must exist and it it must be invoked with a command like parameter

```
meson setup builddir --cross-file path/to/cross_file.txt
```

Mesons approach to cross-compilation is that both cross-compiled targets and native targets can be specified in the same build. This means that, e.g. unit tests can be run on the build machine while also compiling for an embedded target. This is very different from CMake, where two different build output directories must be configured.

When invoking a build in Meson with or without specifying a cross-file, the built-in OS toolchain is used for the native build. Like there are cross-files, native-files can also be created to specify which toolchain to use for the native part of the build.

When layering cross-files in Meson, the cross-files must all be invoked in the command line, when configuring the project. when specifying multiple cross-file or native-file arguments, duplicate values from the previous file are overridden by those in the following file.

```
meson setup builddir --cross-file arm.txt --cross-file cortex-m3.txt
```

Like CMake, Meson has a variable to see if a build is a cross- or a native build. The variable is part of the `meson` object called `meson.is_cross_build()` and it returns true if the current build is a cross build.

req#1.6.1

"The system must be capable of running on a Windows x86_64 machine"

Common

Both systems are capable of running on a native Windows machine.

req#1.6.2

"The system must be able to cross-compiling for the following host machines"

Common

req#1.6.2.1 - req#1.6.2.4: *"nRF52810, nRF52832, STM32F03x, STM32G03"*

Since neither CMake nor Meson are responsible for compiling, linking, etc. but only points to the executables handling these steps, an arbitrary number of cross-compilation toolchains can be defined to encompass all the listed processors.

CMake

CMake makes use of toolchain files in the form of `my_toolchain_file.cmake`. The built-in toolchain variables are set to the desired value for the system to use.

```
set(CPU_NAME    STM32F03x)
# ...
set(CMAKE_C_COMPILER    arm-none-eabi-gcc)
# ...
```

In addition to toolchain executables, CMake also provides standard variables for compiler, linker, and VFP (*Vector floating point*) flags, which can be set to target a specific architecture.

It is possible to define a custom root path in the toolchain file by setting a list of root paths in `CMAKE_FIND_ROOT_PATH`. If this variable is defined and used, CMake uses the paths in this list as alternative roots to find filesystem items with functions like `find_library`, `find_path` etc.

The layering of toolchain files to gather common components between different target architectures can be done and is handled in CMake with the use of `include(path/to/toolchain.cmake)`.

Meson

Meson uses cross-files to define specific components of the toolchain. The cross-files are in the form of `my_cross-file.txt`. The files are divided into different sections; `binaries`, `properties`, `built-in options`, `host_machine`, and `target_machine`.

`[binaries]` contains the compile, archive and strip executables, provided they can be found in the environment Path. Alternatively, a full path to the executable can be set.

`[properties]` in this section, size and alignment settings can be set, and this is e.g. also where the objcopy toolchain component can be set.

`[built-in options]` contains compiler arguments, link arguments etc. can be added.

`[host_machine]` and/or `[target_machine]` must be defined in the cross-file, if one of these are not present, the build will be treated as a native build.

`[host/target_machine]` defines the system, cpu family, cpu, endian amongst others.

Layering of cross-files is also possible in Meson. This is done by invoking several cross-files in the command line. further description in the description of req#1.5

req#1.7

"The system must be able to fetch content from an SVN server and from GIT"

Common

Both systems fulfill this requirement.

CMake

CMake has two different methods of fetching content from SVN or GIT, the `ExternalProject` module, and the `FetchContent` module.

The first module can be used to isolate the external project from the main project. It can use a different toolchain, target a different platform, etc. This can be beneficial in some cases, but it also means that the information from the external project must be provided manually. Content fetched with `ExternalProject` downloads the content during build time, making it difficult to build the external project with the same configurations as the main project.

`FetchContent` was introduced in CMake 3.11 and uses the `ExternalProject` module internally to set up a sub-build which downloads and updates the external content. The big difference between the two is that `FetchContent` does all of this during configuration. Because of this, the content is available earlier than with `ExternalProject`, and it is possible to bring the project's sources directly into the main build.

SVN

```
include(FetchContent)

FetchContent_Declare(
    content_name
    SVN_REPOSITORY  <URL>
    SVN_REVISION    <RVN>
)
```

GIT

```
include(FetchContent)

FetchContent_Declare(
    content_name
    GIT_REPOSITORY  <URL>
    GIT_TAG         <TAG>
)
```

Meson

Meson uses its own system `Wrap` for fetching and managing external content. In Meson, external content is fetched under the concept name *subprojects*. They are a way of nesting one Meson project inside another. subprojects can be used in a variety of different ways

- They can be included as a submodule within the source tree, and reference the source files directly in the build
- Subprojects libraries can be compiled in their native setup, and be included as pre-built files and headers in the source tree.
- Precompiled binaries and libraries can be used with Meson's built-in dependency lookup method (e.g. to find pkg-config files).
- The built-in dependency lookup method can be used to find CMake modules.

To get Meson to download the subproject, a wrap file must be specified. By doing this, Meson will automatically download and extract it if it is used in the build. Wrap files are specified in the form `<project_name>.wrap`, and must be placed in the `subprojects` directory. Meson supports four kinds of wraps; `wrap-file`, `wrap-git`, `wrap-hg`, and `wrap-svn`.

A wrap file has a single header that contains the type of wrap, followed by properties that describe how to obtain the sources, validate them, and modify them if needed.

SVN

```
[wrap-svn]
url = http://svn/path/to/dir
revision = revision_number
```

GIT

```
[wrap-git]
url = http://git/path/to/dir
revision = revision_number
```

Additional configuration properties can be given to both wrap-files like

- `patch_url`: download url to retrieve an optional overlay archive.
- `patch_fallback_url`: fallback url if `patch_url` fails.
- `patch_directory`: overlay directory. Must be placed in `subprojects/packagefiles`
- `method`: specifies the build system used by the subproject (default meson).
  - `method = meson|cmake|cargo`

`patch_directory` can be used to provide a build file to the downloaded subproject, if it doesn't inherently have one.

Git-specific properties can be configured in the wrap-git file such as `depth` and `push-url`.

req#1.8.1

"The system must have the capability to build different project versions based on a specified revision number"

Common

None of the systems have built-in functionality to build an entire project based on given revisions or tags. A top-level shim file can be made to interact with the build system in a way that makes it possible to build a project with a given revision or tag.

req#1.8.2

"The system must enable the selection of module versions based on a specified revision number"

Common

Both systems fulfill this requirement. see requirement 1.7 description.

req#1.9

"The system must be allowed for commercial use, preferably at no cost"

Common

Both systems are allowed for commercial use, at no extra cost.

CMake

CMake is distributed under the OSI-approved BSD 3-clause license [10]. This license is a permissive open-source license that allows for both commercial and non-commercial use.

Meson

Meson is licensed under the Apache 2 license [17]. This is a permissive open-source license that allows for both commercial and non-commercial use.

2.6.2 Non-functional Requirement Comparison

To test some of the non-functional requirements, a test project has been created. The tests were carried out with exactly the same project and configurations on three different build machines. The test project consists of 11 separate .c source files and 3 .h header files. The contents of the source files are simple mathematical functions, not serving a particular purpose other than adding a set of operations, so the compiler will not exclude any files during compilation. A main.c file is created to utilize all the functions in the different source files. The test project files

can be found in Appendix B Test Project and at https://github.com/Haugenau20/cmake_meson_thesis.

The project is cross-compiled and targets the MCU architecture family `STM32F103`. Both projects use built-in options to select build type, and are both set to `Debug` internally.

All three machines used the following versions of build generators, back-end build system and toolchain utilities:

- CMake 3.27.7
- Meson 1.2.3
- Ninja 1.11.1
- Make 4.4.1
- gcc 13.1.0
- arm-none-eabi-gcc 10.3-2021.10
- arm-none-eabi-objcopy 2.36.1.20210621

CMake test setup

The CMake setup consists of several `CMakeLists.txt` files. In the root directory, the project is set up and a second CMakeLists, located in the src directory of the project, is added using the function `add_subdirectory(src)`.

Also in the root of the project, the executable called `sample_app` is defined. It targets `main.c` and is linked to the two libraries `c_lib` and `new_c_lib`, both defined in the CMakeLists in the `src` subdirectory.

```
add_executable(sample_app)
add_sources(sample_app PRIVATE ${CMAKE_SOURCE_DIR}/src/main.c)
target_link_libraries(sample_app PRIVATE
    c_lib
    new_c_lib
)
```

Code Block 2.6: Defining the Executable

The libraries are created in the subdirectory `src/CMakeLists.txt`. `c_lib` and `new_c_lib` are set up in the same way, with the only difference being the target sources.

```
add_library(c_lib STATIC)
target_sources(c_lib PRIVATE
    a.c
    ...
)
target_link_options(c_lib PUBLIC "-nostartfiles")
```

```
target_include_directories(c_lib PUBLIC ${PROJECT_SOURCE_DIR}/lib)
```

Code Block 2.7: Creating Libraries and Adding Sources

Custom functions and commands have been made to convert the output from `ELF` format to `.bin` and `.hex`, and to create a memory map file.

CMake has been invoked with the following commands:

- Makefile backend

```
cmake -B buildresults -G "Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE=
    "cmake\toolchains\cross\STM32F103VBIx.cmake" > output.txt
```

- Ninja backend

```
cmake -B buildresults -G Ninja -DCMAKE_TOOLCHAIN_FILE=
    "cmake\toolchains\cross\STM32F103VBIx.cmake" > output.txt
```

- Invoke backend build tool

```
cmake --build buildresults --verbose >> ouptut.txt
```

Meson test setup

The Meson setup is created using two `meson.build` files, one in the root of the project, and one in the `src` subdirectory.

In the root `meson.build` file, the project is defined with a name, version number, and default options. In addition to defining the project, the src subdirectory is included. All other definitions are handled in the src meson.build file.

```
subdir('src')
```

In the `src` directory, the files, include directories, and flags are listed, before the static library `c_lib` is created, and a dependency variable `c_dep` is created.

```
c_files = files(
    'a.c',
    ...
)
c_include = include_directories('../lib')
c_link_flags = ['-nostartfiles']
c_lib = static_library(
    'c',
    c_files,
    include_directories: c_include,
)
c_dep = declare_dependency(
```

```
    include_directories: [
        c_include,
    ],
    link_args: c_link_flags,
    link_with: [c_lib, new_c_lib],
)
```

Code Block 2.8: Creating Source File Dependency c_dep

Since Meson builds both for the build machine and the host machine in the same build, all linker script-related flags, dependencies, custom targets, and the memory map creation are wrapped inside an `if meson.is_cross_build()` statement.

Meson has been invoked with the following commands:

- Ninja backend

```
meson setup buildresults --cross-file build/cross/arm.txt
    --cross-file build/cross/STM32F103VBIx.txt > xxx_output.txt
```

- Invoke Ninja build

```
meson compile -C buildresults --verbose >> xxx_output.txt
```

req#2.1.1

"The system must ensure that the same building steps are executed consistently each time the build is run, regardless of the build machine or invoker"

Common

Both systems show similar behavior across all three devices when using Ninja as the back-end build system. The two systems identify the compiler specified in the corresponding toolchain file, configure the project, and generate the Ninja build files.

Since Ninja executes the build in parallel by default, the steps are not necessarily the same every time. Not even when running it several times on the same build machine. It does, however, follow the same general steps every time it is run.

1. The soufce files are compiled into object files.
2. Objects are linked to their respective libraries.
    - If not done before the previous step, `main.c` is compiled into an object file.
3. The executable `sample_app` is linked with all its dependencies.
4. The functions / custom commands are executed to create the `.hex` and `.bin` files.

The fact that the individual build steps are not completed in the same order every time does make it more difficult to predict how every build is run, down to which object file is created before the other. The consequences of not compiling in the exact same order every time are minimal. This is because each source file is compiled into objects individually, without regard to the other source files. The most important thing is that all necessary object files are created before the toolchain archiver creates the associated libraries.

The build output files of each test on both systems demonstrate this. They can be found in the terminal output folders of the GIT repository at https://github.com/Haugenau20/cmake_meson_thesis.

If `Ninja` is forced to execute a serial build, the execution order is the same every time, also when it is run on different build machines.

CMake

CMake also has the option to generate Makefiles and use it as its back-end build system. When invoking the build with the Makefile backend, the order of steps run on each build machine is exactly the same every time it is run. It first compiles the source files for `new_c_lib`, links to the static library, and builds `new_c_lib`. The same is then done for `c_lib`, before the `main.c` source file is compiled and `sample_app` is linked and built. This is true because Makefiles run in serial by default.

It is possible to run the MakeFile build in parallel, and when done, the build expresses the same behavior as running Ninja in parallel.

req#2.2.1

"The system must be capable of generating an identical hex output file when the same configuration is used"

Common

Using the test setup, a `sample_app.hex` output was created from each of the setups.

The common for both systems is that the 3 individual hex files created from each build machine match 100%. The hex files produced when invoking CMake and invoking Meson are not a match, when compared to each other. However, they are very similar in size, only differing by 58 bytes. The CMake made `sample_app.hex` has size `10.467 bytes` and the Meson has size `10.525 bytes`.

The difference in output between the two systems can be due to different default compile and link flags set by the systems, and the way the system integrates with the back-end build system.

CMake

The hex output files generated when using Makefile or Ninja as back-end match each other 100%

req#2.3.1

"The system must produce an updated output, when changes are made within the source tree"

Common

Both systems rely mainly on the back-end build system to detect and rerun any necessary compile or build steps. Reconfiguration and generation of the back-end build files are handled by the respective build system.

CMake

If a change has been made to a `CMakeLists.txt` file, rerunning the original cmake command at the command line will invoke new configuration and generation steps. The same functionality can be achieved by invoking Ninja from the build folder in the command line.

Meson

Meson will not reconfigure itself if a change is made in one or more `meson.build` files, if it is invoked with the same setup command used at the beginning. If a reconfiguration is desired, it must be added to the command line with `--reconfigure`.

In Meson's compile command, `meson compile`, is invoked, it will invoke Ninja (or another configured back-end), which will trigger a reconfiguration of meson.

req#2.4.1

"The system should have widespread industry usage"

CMake

CMake is widely used in industry and, according to Kitware, is the de facto standard for building c++ code today [7]. A few companies that have adopted and use CMake are Garmin, AMD, Epic Games, Blender, and others. [27] [4]

Meson

Meson was developed and released later than CMake, and it does not have the same foothold in the market as CMake. Meson offers a list of projects using the Meson build system, which includes projects like GIMP, PicoLibc, MiracleCast, etc. [18]

req#2.4.2

"The system should have comprehensive documentation"

CMake

Kitware provides a comprehensive reference documentation website that includes everything from tutorials to descriptions of every function and variable used in CMake. Different versions of the documentation page can be selected to easily find the correct documentation for the version of CMake used.

Meson

Meson provides extended documentation on everything contained in the build system, all the way from *"The Absolute Beginner's Guide to Installing and Using Meson"* to a full-fledged manual, reference table, and FAQ page. [19]

req#2.4.3

"The system should have strong community support"

CMake

Because CMake has been around since the early 2000s, there is plenty of community support available on the internet. Kitware provides its own Discourse Forum, where users can post questions and get help from community members and also from Kitware employees.

They also provide a CMake community Wiki page, where registered users can request access to contribute to new pages or update existing ones. [3]

Meson

Meson provides a web chat through a web interface or is accessible through OFTC (*The Open and Free Technology Community*). They have a mailing list and a discussion section on their Meson GitHub repository. Meson also provides recommendations on how to organize and format the `meson.build` files. [5]

req#2.5.1

"The system should provide warning and error messages that are helpful for locating errors in the system"

Common

For debugging the project, both systems rely on the back-end build system and the toolchain used.

Both systems rely heavily on message functions that are available for the system developer to tailor for the specific project.

CMake

CMake offers a command to log arbitrary text using its `message()` function. The messages can have several different levels from `FATAL_ERROR`, `WARNING`, and `DEPRECATION` to `NOTICE`, `STATUS`, and `DEBUG`. Depending on the type of message, it is printed to `stdout` (*standard output*) or `stderr` (*standard error*).

CMake has a set of check messages to be used in combination with if-else statements to notify if a custom operation has succeeded or failed. The string passed to `CHECK_PASS` and `CHECK_FAIL` will be appended to the `CHECK_START` message and will be printed again.

```
message(CHECK_START "Checking support")
if(var1)
    message(CHECK_PASS "supported")
else()
    message(CHECK_FAIL "not supported")
endif()
```

Code Block 2.9: CMake Check Messages

```
-- Checking support
-- Checking support - supported
```

CMake offers a variable to specify color diagnostics behavior for compilers, without the developer needing to know which specific flag(s) is supported or if the compiler even supports color diagnostics. The variable is called `CMAKE_COLOR_DIAGNOSTICS`, and is only supported in cmake 3.24 and later. Setting the variable to true will enable color output for compilers that support it.

Meson

Like CMake, Meson has a function to log arbitrary messages, but does so with several different functions.

- message()
- warning()
- error()
- debug()

`message()` will print its argument to `stdout` and will be displayed with a `"Message:"` prefix.

`warning()` also prints its argument to `stdout`, and is displayed with the path to where it is called along with a colored `"WARNING:"` prefix.

`error()` will print its arguement to `stderr` and halts the build process. The error messages is printed with a path to the functions call and prefixed with a colored `"ERROR: Problem encountered:"`.

`debug()` won't print its argument, but write it to the meson build log, placed in the configured build folder.

Meson also has an `assert()` function, which takes a boolean argument. If it is false, it will stop the build process and print a custom error message.

req#2.6.1

"The system should not require an extensive setup process when set up with a project template"

Common

With a reusable project template, a project setup consists mainly of specifying source file names and paths to the system and providing the system with any project-specific compile or linking flags. When the template is set up, it is not expected that creating new projects will be an extensive or time-consuming task.

req#2.6.2

"The system should be able to invoke a specific build using a single command"

Common

Both systems can be invoked directly from the command line, and depending on the type of build, might require additional command line arguments to setup the build in the preferred way. This may lead to human errors if a build depends on the developer to input the correct arguments when setting up the build. An easier method would be to create a top-level shim file that is used to interact with the system.

This is possible in a similar manner for both systems.

req#2.6.3

"The system should provide an easy-to-learn interface for invoking builds"

Common

The recommended approach for both systems is to create a top-level shim file using Make. This can help narrow down the number of command arguments that the developer has to input when invoking a specific build. Creating a shim file for interacting with the system will allow one call from the Makefile to call several build system-specific commands, and also necessary commands not affiliated with the specific system, if needed.

Creating a standard Makefile that can be used across all projects in the company will allow developers to learn a single interface independent of the specific project. This will create more time to focus on the project than on how to build the project.

req#2.6.4

"The system should have an easy-to-understand syntax for developers not maintaining it on a day-to-day basis"

CMake

Because CMake has been around since 2001, it has grown a lot and is a large and powerful build system. With many different versions of the system released and large steps between some versions (e.g. from CMake v2 to v3), it is capable of doing almost anything desired from building projects to testing and packaging them. From a syntax-user-friendly perspective, this means that the learning curve is quite steep. Most built-in functions are self-explanatory as to what their purpose is, and depending on the used editor environment, help text is provided for input and output parameters.

Meson

The Meson syntax has been developed to keep it as simple as possible. The language is strongly typed, meaning that no object is ever converted to another "under the hood". It is also dynamically typed, meaning that variables have no visible type.

Meson is implemented in Python and is built in such a way that it does not have any dependencies outside of the Python basic library. It has been designed so that the implementation language is never exposed in the build definitions. This means that the user of Meson does not need to have any underlying knowledge of Python or any other language if Meson is ever reimplemented in another programming language.

Meson's build in function `"dep declare_dependency(...)"` makes integrating modules fetched from e.g. SVN very easy. The returned `dep` object can be passed to the root `meson.build` file specifying how the module should be used.

req#2.7.1

"The system must not exhibit noticeable performance lag compared to the currently used Keil system"

Common

Depending on the scale and complexity of the project, performance can be significantly affected when using Make or Ninja as the back-end build system. The build machine used for the build

will also impact the performance; in general, the more CPU cores and a faster clock speed, the faster the build.

A test was run 10 times with the same configuration on each build machine, and an average configuration and build time was calculated for each. The measured time is the build system generator (CMake, Meson) configuration and generation time plus the time it takes for the toolchain to compile and link the project.

The tests were measured with the `Measure-Command{}` function in a Powershell environment:

```
$executionTime = Measure-Command{<build_system_commands>}
$executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append
```

It was decided not to set up the test project in *µVision Keil* due to the time available to set up the project versus the output value. It is not expected that the output will be noticeably faster or slower than the tested build systems. In the following chapters, a full-scale performance test and comparison with the demonstration software will be conducted.

CMake

CMake has the ability to use Make and Ninja as a back-end build system.

Using the test project, different performance measurements were made when using Make, Make with parallel jobs, and Ninja with parallel jobs.

Backend	**Avg. time [s]**	**Min. time [s]**	**Max. time [s]**
Make	10.565	9.762	11.088
Make (–parallel 8)	7.847	6.797	8.332
Ninja	5.085	4.553	5.575

Table 2.1: Work laptop perfomance test, CMake

Performance output from the other build machines can be found in Appendix B.3

Meson

Meson does not have the ability to generate Makefiles, so tests were only carried out using Ninja with parallel jobs.

Backend	**Avg. time [s]**	**Min. time [s]**	**Max. time [s]**
Ninja	7.844	7.148	8.648

Table 2.2: Work laptop perfomance test, Meson

Performance output from the other build machines can be found in Appendix B.5

req#2.8.1

"The system must be developed and maintained by a team with a consistent and reliable history"

Common

Both systems are open source platforms and most contributions to the systems come from outside of the core maintainers. Both systems have a hierarchy established where anyone can report bugs and suggest changes, but only a smaller team of trusted maintainers decides what and how anything should be implemented.

CMake

CMake is maintained and supported by **Kitware** and developed in collaboration with a productive community of contributors. The core members of the maintaining team either work for Kitware or have a strong affiliation with the company.

Meson

Meson does not have the backing of a large company, but was developed by a single person as an open source project. Since its creation, it has grown larger and now has a core team of maintainers attached to the project. The creator of the system is still one of the main maintainers of the system, including a fair number of other developers. A list of contributors can be found on the project's GitHub page. [6]

req#2.8.2

"The system should be maintained by a sufficient number of developers"

Common

It is difficult to pinpoint the exact number of core contributors for systems, since they are both open source platforms.

CMake

Because of CMake's size and widespread usage in the industry, it is assumed that a sufficient number of developers are maintaining the system.

Meson

Meson has a long list of contributors (*approximately 100*), with 11 developers who have committed changes more than a hundred times. Most of these are still actively maintaining the system, based on the commit activity and amount. [6]

req#2.8.3

"The system should have the ability to switch compiler version"

Common

As both systems are not handling compiling, linking, etc. themselves, this is possible by pointing to the toolchain that is wanted for the given project. An infinite number of toolchain or cross-files can be created for both build systems.

req#2.8.4

"The system should allow users to suggest new features, report bugs, and provide feedback"

CMake

With CMake it is possible to file bugs, suggest new features and provide feedback to the system through Kitware's CMake git page in the *issues* section. When proposing new features or reporting bugs, a detailed instruction must be included, and the system maintainers will review it to decide if it is something to look into or not.

Meson

Anyone can report bugs or suggest new features for the Meson build system. As opposed to the larger team of developers at CMake, it is to a higher degree welcomed at Meson that the person submitting a new feature idea is also the one who comes with the implementation code itself.

req#2.8.5

"The system must be used by a defined user base in the industry"

Common

Both systems are used to some extent in the industry, CMake more than Meson. To see examples of projects using the two systems, see the description of the requirement 2.4.1.

req#2.9.1

"The system should be supported in IDE's with syntax highlighting, code completion and hover information"

CMake

req#2.9.1.1 *"Support in VS code"*

VS Code offers strong support for CMake with installed and enabled extensions of **CMake Tools** and **CMake Language Support**. VS Code provides auto-configuration that can be set whenever a `CMakeLists.txt` file is saved, and upon opening a project. [22][28]

req#2.9.1.2 *"Support in Eclipse"*

Eclipse offers support for CMake in ways similar to VS Code when the plugin **CMake Editor** is installed. The plugin provides syntax highlighting, basic hover support, and code completion for CMake's own functions. [29]

req#2.9.1.3 *"Support in Keil"*

Keil currently does not offer any support for editing CMake files.

Meson

req#2.9.1.1 *"Support in VS code"*

Meson has support in VS code with the extension **Meson**. This extension allows for syntax highlighting, code completion, and hover information. [21]

req#2.9.1.2 *"Support in Eclipse"*

Eclipse offers limited support for Meson. It can help set up a simple project and provides syntax highlighting. It does not provide code completion or hover information.

req#2.9.1.3 *"Support in Keil"*

Keil currently does not offer any support for editing Meson files.

2.6.3 Evaluation Matrix

#	Requirement	CMake	Meson
1.1.1	Convert source code to binary code	✓	✓
1.1.2	Produce hex output	With toolchain conversion executable	With toolchain conversion executable
1.1.3	Generate statistics	Static analysis. 3rd party integration possible	Static and coverage analysis. Code sanitizer. 3rd party integration possible
1.2	Different build types	✓ Custom build types possible	✓ Custom build types possible
1.3	Out-of-source build	✓	✓
1.4	Incremental building	✓	✓
1.5	Invoke cross-compiling	✓	✓
1.6.1	Works on x86_64	✓	✓
1.6.2	Several cross-compile toolchains	✓ Specified in individual toolchain files	✓ Specified in individual cross-files
1.7	Fetch from SVN/GIT	✓	✓
1.8.1	Project revision	With custom implementation (shim file, scripts, etc.)	With custom implementation (shim file, scripts, etc.)
1.8.2	Module revision	✓ Declared in module fetch functions	✓ Declared in subproject .wrap files
1.9	Commercial use	✓	✓
2.1.1	Repeatable	Make serial: 100% Make parallel: ∼100% Ninja serial: 100% Ninja parallel: ∼100%	Ninja serial: 100% Ninja parallel: ∼100%
2.2.1	Reproducible hex output	✓	✓
2.3.1	Output reflects the source tree	Custom dependency on linker-scripts needed	✓

Continued on next page

Table 2.3 – continued from previous page

#	Requirement	CMake	Meson
2.4.1	Used in the industry	To a large degree	To a small degree
2.4.2	Comprehensive documentation	✓	✓
2.4.3	Strong community support	✓ Strongest	✓
2.5.1	Means to debug	✓	✓
2.6.1	Easy setup w. template	✓	✓
2.6.2	Invoke w. single command	With shim-file implementation	With shim-file implementation
2.6.3	Easy to learn	With shim-file implementation	With shim-file implementation
2.6.4	Syntax	Steep learning curve	Slightly less steep learning curve, python-like syntax
2.7.1	Performance	TBD Expected negligible difference	TBD Expected negligible difference
2.8.1	Maintained by	Kitware + Community	Community
2.8.2	Sufficient contributors	✓	100 contributors *
2.8.3	Able to include new compilers	✓	✓
2.8.4	Suggest new features, report bugs, etc.	✓ Longer process with more mature system	✓ Implementation suggestions appreciated
2.8.5	User base	Large user base	Small user base
2.9.1.1	VS Code support	✓	✓
2.9.1.2	Eclipse support	✓	Limited
2.9.1.3	Keil support	No support	No support

* Contributors as of 10-26-2023 [6]

Table 2.3: Evaluation Matrix

As can be seen in Table 2.3, CMake and Meson fulfill to some extent all the listed requirements. They fulfill them in their own way, which makes some of the requirements more interesting than others when one system must be chosen for further development.

The non-functional requirements are where the differences are greatest because they each contain more nuanced answers than capability requirements. Since both systems fulfill all of the requirements, a few have been selected, where the difference in how they fulfill the given requirement is greatest.

- req#2.4.1 *"Widespread industry usage"*
- req#2.4.3 *"Strong community support"*
- req#2.6.4 *"The system should have an easy-to-understand syntax for developers not maintaining it on a day-to-day basis"*
- req#2.8.1 *"Maintained by a team with a reliable history"*

CMake has a strong lead in all the four above requirements. The four requirements go well together and support each other when looking at how development and support will be in the future. The largest of the four is that it is developed and maintained with the backing of the large international company (*Kitware*), as opposed to Meson with the backing mainly of the inventor and a handful of stable contributors. Because of this, the risk of either project being discontinued is much greater for Meson than for CMake.

2.7 Choice

The system comparison section of the analysis was sent to DLSW on October 31 for review prior to a decision-making meeting on November 21. During the decision-making meeting, the requirement summary matrix was presented to the manager, architects, and selected developers at DLSW. The focus of the meeting was the main differences between the two systems, as mentioned above in 2.6.3 Evaluation Matrix. The participants had the opportunity to ask more specific questions to the two systems to get a better understanding and a foundation on which to base their choice.

The ease of use and syntax of the two systems were compared and discussed. There was a general consensus that Meson has a better and more descriptive syntax than CMake. It would be easier to work with and maintain Meson, if not working with the system on a day-to-day basis. However, it was also discussed that the difference between the two was not of such a magnitude that this alone was able to sway the decision greatly over to Meson. Whilst Meson has an easier to understand syntax, members of DLSW already has some experience with CMake through their unit test setup. Having some experience with CMake already makes up for the more difficult syntax in CMake compared to learning a whole new system.

The largest contributing factor for choosing CMake was agreed to be the amount of contributors

it has and the fact that they are developed by and backed by a large company. Having this backing whilst also being an open-source project, where it is possible to contribute by reporting bugs and suggesting new features, sets CMake in a much better position when looking at future perspectives.

One of the main pitfalls of Meson was due to the small number of contributors. With a smaller number of contributors, the risk of the project being shut down or discontinued is greater. It is greater because the risk of the handful of contributors deciding to leave the project, and not get replaced, is much larger than with CMake.

At the end of the meeting, it was decided that CMake should serve as the system to develop the demonstration software. The participants based this decision mainly on the widespread use of CMake in the industry, the strong community support, and the fact that it is maintained by a large team of contributors, one of them being the company *Kitware*.

Chapter 3

Demonstrator: Design

This chapter outlines the design of the demonstration software. It considers the requirements agreed upon by DLSW in Chapter 2, and the list of detailed objectives in Chapter 1.

To test the CMake build system on a larger scale, a current project at DLSW has been selected. The project has been selected in agreement with the LINAK supervisor and is LINAK's DF3 (*Desk Frame 3*) system.

This project has been chosen due to it's modest size, it's use of DLSW's CLEAR SW platform, and because it is compiled with Arm Compiler For Embedded (Arm compiler 6).

The goal of the demonstration software is to migrate the current build process from *µVision Keil* to CMake. The demonstration software will showcase the following main features:

- Capability to build a current DLSW project.
- Generate a usable output using the Arm compiler 6 toolchain in *.hex format.
- Include modules from DLSW's CLEAR platform.
- Be independent of the use of a specific IDE.
- Contain options for different compiler settings.

3.1 Demonstrator Overview

The setup of the demonstration software follows a general structure for the files and directories. As there is no current structure for setting up a CMake project at DLSW, the following is a suggestion of how the internal structure can be designed. The structure should facilitate an easy understanding of where to find different functions and custom settings if reviewing the build setup. The names of directories and files should be as descriptive as possible without being overly verbose. Functions, lists, etc. should be separated as much as possible to improve

readability, maintainability, and provide a better overview of common and project-specific elements for future development.

To separate common and project-specific files, all common functions and toolchain files are placed in a separate directory called *cmake*. All project-specific files, like scatter files, build flags, etc. are placed in the root directory or in their own subdirectory, for increased readability.

A general directory tree can be seen in Figure 3.1. The final directory tree of the demonstration project can be found in Appendix D.7

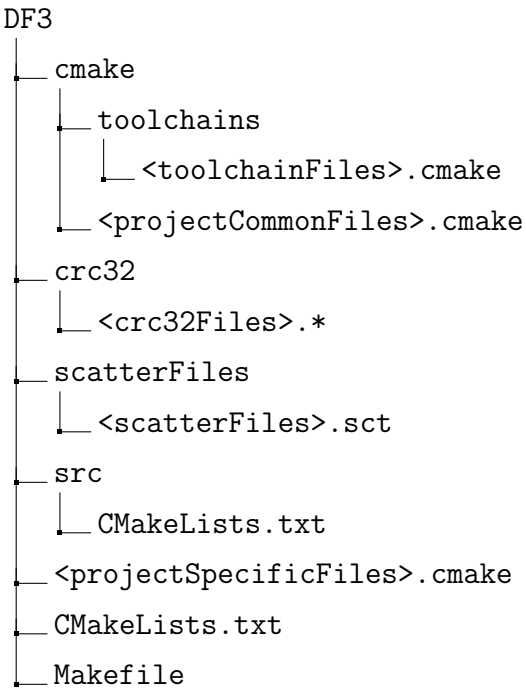


Figure 3.1: General Project Directory Structure

3.2 Component Design

To provide better readability, maintainability, and reduce redundancy in work, each project component must implement a specific functionality, defined as the responsibility of the component. The responsibilities of the main component in the project are defined below.

3.2.1 Toolchain Files

A set of toolchain files must be created, and they are responsible for defining the MCU, architecture, and toolchain-specific options to allow cross-compilation. The toolchain files must be split up to reduce redundancy of work if a new MCU is added to the project at a later stage. For this project, the MCU `STM32G03x` is used, and a toolchain file must be created to set all

MCU-specific options, another file to define architecture-specific options, and a third file to define the toolchain executables. They should be layered on top of each other in such a way that the variables defined in the MCU file are passed to the architecture file and from there to the toolchain-specific file.

3.2.2 Project Common Files

Common files for the project must consist of a set of files defining support functions for the build. Functionality needed to make this project possible includes

- Functionality to fetch content from the SVN server.
- Functionality to create *.hex files and add a checksum value to the output file.
- Add a dependency from the output executable to a build-type-specific scatter file if the toolchain defines it.
- Functionality to add a list of compile and linking flags to the build.

3.2.3 Main CMakelists

The main CMakeLists.txt file must be in the project root directory. This file will be responsible for the configuration of the project, including all variables, functions, and settings used in the project. It must do so by calling the CMake `project()` function to set the project name, version, and languages used. The main CMakeLists.txt is responsible for handling all SVN fetching, using the functionality from the project common files. The modules to fetch, the module's version, and the desired revision number must all be provided from this file and passed to the implemented functionality.

After everything is set up, it is responsible for calling and processing the source CMakeLists.txt.

3.2.4 Source CMakeLists

The second CMakeLists.txt file in the project will be located in the `src` directory. It will use all files fetched in the root CMakeLists and is responsible for defining the output executable of the project. The CMake function `add_executable` must be called to define the executable name. It is responsible for providing all the relevant source and header files needed for compilation and linking to create the target executable.

3.3 Project Interface

To create an easier interface for developers, a Makefile is created at the root of the project. This interface file will be responsible for setting up the following targets and options:

- A target to initiate CMake configuration.

- A target to initiate compilation and build.
- A target to delete build artifacts.
- A target to purge the build directory.
- A help target to print out information on how to interact with the interface.
- Relevant options and means to pass them to the configuration and build.

3.4 Summary

This chapter outlined how the project will be set up by describing the general project structure and describing the responsibilities of the main components. The details covered in this chapter will be carried over and used in the next chapter Demonstrator: Implementation, where the functionality of the project is implemented and described in detail.

Chapter 4

Demonstrator: Implementation

This chapter looks at the implementation of the design decisions made in chapter 3. This includes a detailed walk-through of how the project is set up and of all the individual files. The implementation is carried out to achieve the objectives set in Detailed objectives, which will be verified and discussed in chapter 5.

4.1 Fetching From SVN

The CMake setup of the DF3 project begins with the fetching of all the source files for the project. All source files used for the project are currently on LINAK's SVN server and need to be fetched to the build machine before they can be included in the project. A set of macros is created to fetch the modules, drivers, etc.

DF3/cmake/fetchFunctions.cmake

This cmake file defines several macros to fetch content from the LINAK SVN server. To fetch content, the third party dependency manager CPM is used. CPM is a cross-platform CMake script that adds dependency management capabilities to CMake. It is built as a wrapper around CMake's `FetchContent` module that adds version control, caching, API, and more. [15]

The most used macro is `fetch_clear_module()`;

```
macro(fetch_clear_module MODULE_NAME VERSION REVISION
  message(STATUS "Fetching CLEAR module: ${MODULE_NAME}:${VERSION}
                  @ ${REVISION}")
  CPMAddPackage(
    NAME ${MODULE_NAME}
    SVN_REPOSITORY ${SVN_REPO}/CLEAR/mod/${MODULE_NAME}/${VERSION}/lib
    DOWNLOAD_ONLY ON
    SVN_REVISION -r${REVISION}
endmacro()
```

Code Block 4.1: CMake Macro to Fetch a CLEAR Module

This example shows an implementation of how a CLEAR module can be fetched, where `MODULE_NAME` is the name of the module to be fetched, `VERSION` is the version to be fetched (e.g. *trunk*, *branch/<branch_name>*), and `REVISION` is the revision at which the module should be fetched.

A status message is sent to the user through the command-line interface.

Then the CPM function `CPMAddPackage` is called, passing the input arguments. The function fetches the content in the repository path, populates it so that it is available to the project, and sets up several different variables to access the content. The variable `<module_name>_ADDED` is set to true if the fetching succeeds. When `DOWNLOAD_ONLY` is set to `ON`, CMake will not search for or include any CMakeLists.txt that might be in the root directory of the fetched content.

The same approach is used to fetch drivers, common files, external libraries, and platform-specific files:

`fetch_clear_driver(TYPE VERSION REVISION)`: Since different MCU's are used at DLSW, a function has been made to be able to specify which MCU specific driver should be fetched. `TYPE` is the argument to set to choose between different MCU's.

`fetch_clear_common(MODULE_NAME VERSION REVISION)`: Much like the macro to fetch modules, this one fetches common files from CLEAR.

`fetch_extlib(EXTLIB_NAME EXTLIB_VERSION REVISION)`: Several external libraries are used in the project and can be fetched with this macro. `EXTLIB_NAME` specifies which external library to fetch (e.g. *boost*).

`fetch_df3_platform_files(TYPE VERSION REVISION)`: This macro is made specifically for the demonstration project and is used to fetch platform specific files for DF3 (e.g. *configuration* files).

`fetch_df3_app_files(VERSION)` Like the platform files, the application file macro is also made for the demonstration project to fetch application specific files for DF3 (e.g. *appMain.c*).

4.2 Custom Functions

Spread over several different `*.cmake` files, there are custom functions to overwrite built-in functionality, check compile flags, and convert output.

DF3/cmake/addExecutableWithScatterFile.cmake

In this file, a custom function has been made to overwrite CMake's own `add_executable` command.

```
1 function(add_executable TARGET)
2     _add_executable(${TARGET} ${ARGN})
```

```
3     if(SCATTER_FILE_DIR)
4         string(TOLOWER ${CMAKE_BUILD_TYPE} build_type)
5         set_target_property(${TARGET} PROPERTIES
6             LINK_DEPENDS "${SCATTER_FILE_DIR}/${build_type}.sct")
7         target_link_options(${TARGET} PRIVATE
8             "--scatter"
9             "${SCATTER_FILE_DIR}/${build_type}.sct")
10    endif()
11 endfunction()
```

Code Block 4.2: Overwrite add_executable to Include Scatter File and Dependency

The function first passes all input arguments to the original command (Code Block 4.2 line 2). It checks if the variable `SCATTER_FILE_DIR` has been set (set in the toolchain file). If it is, the build type is passed in a lower case version to the CMake function `set_target_property` to add a dependency between the executable and the scatter file. By doing this, if a change is made in the scatter file, CMake knows that it must reconfigure the executable and apply the changes. Lastly, the CMake function `target_link_options` is called to pass the scatter-file inclusion flags to the toolchain linker.

DF3/cmake/checkAndApplyFlags.cmake

This file adds a function to call CMake's built-in command `check_c_compiler_flag()` to check if the passed flag is supported by the toolchain compiler. If supported, it will be added as a compile flag to the specified target. The user-defined function `apply_supported_c_compiler_flag( target scope flag)` can be found in Appendix section D.5

It also adds three functions to add compiler and linker flags to an executable from the flag lists defined in **DF3/buildFlags.cmake**.

```
1 function(add_exe_linker_flags target scope)
2   foreach(flag ${exe_linker_flags})
3     target_link_options(${target} ${scope} ${flag})
4   endforeach()
5 endfunction()
```

Code Block 4.3: Add linker flags from `exe_linker_flags` list

The function traverses the list `exe_linker_flags` and adds all link flag options to the target executable.

The functions `add_exe_compile_flags` and `add_exe_build_type_compile_flags` are also defined in this file. The latter function adds a different list of compiler flags depending on if the build type is *Release* or *Debug*. These functions can be found in Appendix section D.5

DF3/cmake/conversions.cmake

Three functions are defined in this file `convert_to_hex`, `crc32`, and `target_linker_map`

```
function(convert_to_hex TARGET)
  add_custom_command(
    TARGET ${TARGET} POST_BUILD
    COMMAND ${OBJCOPY} --i32combined --output=${TARGET}.hex ${TARGET}.elf
    BYPRODUCTS ${TARGET}.hex)
endfunction()
```

Code Block 4.4: Convert Target to Hex Format

The function adds passes its input to CMake's `add_custom_command` command. The function adds a post-build command for the input `TARGET`. It uses the `OBJCOPY` variable set in the project toolchain file (in this case, Arm's `Fromelf`) and provides the required input parameters.

`crc32()` is also a post build function that takes the output.hex file and passes it as an argument to the CRC (*Cyclic Redundancy Check*) crc32 executable to append a checksum value to it.

The function `target_linker_map` adds link flags to the toolchain linker, telling it to generate a memory map file for the target executable and place it in the build output directory. The full function can be found in Appendix section D.5

4.3 Toolchains

To cross-compilate the project, one or more toolchain files must be defined. The approach in this project is to have separate files and collect as much architecture and MCU specific information to avoid any redundancy if several MCU's should be supported in the future. The toolchain files are layered on top of each other. By doing this, only the MCU-specific toolchain file must be included when invoking CMake.

DF3/cmake/toolchains/STM32G03x.cmake

The MCU-specific toolchain file for `STM32G03x` contains only the definition of a few variables. An include guard is added at the beginning of the file because CMake will include the file several times when configuring. Several includes are not needed and, in some cases, can lead to errors.

```
if(STM32G03x_TOOLCHAIN_INCLUDED)
  return()
endif()
set(STM32G03x_TOOLCHAIN_INCLUDED TRUE)

set(CPU_NAME STM32G03x)
set(SCATTER_FILE_DIR ${CMAKE_SOURCE_DIR}/cmake/scatterFiles)
set(CPU_FLAGS "-DSTM32G030xx -D__MICROLIB")
set(ASM_FLAGS "--pd \"__MICROLIB SETA 1\"")
set(LD_FLAGS "--library_type=microlib --cpu=Cortex-M0+")
include(${CMAKE_CURRENT_LIST_DIR}/cortex-m0plus.cmake)
```

Code Block 4.5: STM32G03x Toolchain

The toolchain file sets a global variable for the CPU name. It defines the scatter file directory and sets variables to contain compile and link flags chosen to be general for this MCU. Layering takes effect at the bottom of this file when the `cortex-m0plus.cmake` file is included and processed.

DF3/cmake/toolchains/cortex-m0plus.cmake

In the more general architecture-specific toolchain file `cortex-m0plus.cmake` a few CMake variables are set to specify the system processor and architecture. Several compile and link flags are appended to those defined in `STM32G03x.cmake`.

```
1 set(CMAKE_SYSTEM_PROCESSOR cortex-m0plus)
2 set(CMAKE_SYSTEM_ARCH armv6-m)
```

Code Block 4.6: Parts of `cortex-m0plus.cmake`

This toolchain file is layered on top of the `armclang.cmake` toolchain file.

DF3/cmake/toolchains/armclang.cmake

In this top toolchain file system configuration variables, toolchain executables, initial compilation and link flags, and specific build time flags are set. The build time flags are set in `overrideBuildTypeSettings.cmake` and must be included directly in this toolchain file.

```
1 set(CMAKE_C_COMPILER     armclang)
2 set(CMAKE_ASM_COMPILER   armasm)
3 set(CMAKE_AR             armar)
4 set(OBJCOPY              fromelf)
```

Code Block 4.7: Toolchain Configuration, `armclang.cmake`

The toolchain executables must be present in the environment path of the build machine, or a full path to the executable must be provided.

```
1 set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)
```

Code Block 4.8: CMake Try Compile Variable, `armclang.cmake`

Setting `CMAKE_TRY_COMPILE_TARGET_TYPE` will affect how the test compiles are run, when CMake is running compile tests in it's configuration step. CMake tests this to make sure that the compiler used is present and works. When set to `STATIC_LIBRARY` a static library will be built as a test. This is useful when cross-compiling because it avoids invoking the linker, which might not work at this point because the necessary flags, scatter file, or linker script inclusion, are not yet defined.

The flags defined in the previous toolchains are passed and appended to the corresponding CMake variables to be included in the build.

```
set(CMAKE_C_FLAGS_INIT
  "${CPU_FLAGS}"
  CACHE
  INTERNAL "Default C compiler flags.")
set(CMAKE_ASM_FLAGS_INIT
  "${ASM_FLAGS}"
  CACHE
  INTERNAL "Default ASM compiler flags.")
set(CMAKE_EXE_LINKER_FLAGS_INIT
  "${LD_FLAGS}"
  CACHE
  INTERNAL "Default linker flags.")
```

Code Block 4.9: CMake Initial Flags, `armclang.cmake`

The flags are set and stored in CMake's cache. The CMake cache is a set of persistent variables that is used to store configuration settings. These values will remain set and will be used on multiple runs within a project build tree. The cached values can be modified through CMake's GUI, via the command line, by reconfiguring the build, or by purging the build result directory and invoking CMake again [8]. The `INTERNAL` option tells CMake that the value is for internal use and will not show up in the variable list if the CMake GUI is used.[1]

DF3/cmake/toolchains/overrideBuildTypeSettings.cmake

CMake has a set of default flags set for different build types, for example, for a release build, the flags "`-O3 -DNDEBUG`". For this project, a different optimization is required and the default flags need to be overwritten.

```
set(CMAKE_C_FLAGS_RELEASE
    "-Oz"
    CACHE
    STRING "Default C compiler release build flags."
)
```

Code Block 4.10: Override Default Build Flags

A more aggressive optimization flag focused on minimizing the size of the generated executable is desired, and the previous value(s) will be overwritten with "`-Oz`", in this example. `CMAKE_C_FLAGS_RELEASE` will be appended to the compiler flag list when `CMAKE_BUILD_TYPE` is set to "`Release`". Like the flags defined and set in `armclang.cmake`, these values are cached, so they are carried over across builds. The option `STRING` makes the variable available for modification in the CMake GUI.

[1] *CMake's GUI has not been further investigated and its use is not included in this project.*

4.4 Build Options

In the root directory of the project, the files `buildFlags.cmake` and `buildOptions.cmake` are located.

DF3/buildFlags.cmake

The build flags that are not related to a specific architecture, MCU, or build type are all defined in this file. They are appended as variables in a string list and are used in the `add_exe_*` functions defined in DF3/cmake/checkAndApplyFlags.cmake.

DF3/buildOptions.cmake

Default options and custom build options are defined in this file. If no value is provided for the variable `CMAKE_BUILD_TYPE`, it will default to an empty string. This behavior is not desired because the build type variable is accessed to set specific flags, scatter file, etc., for the build.

```
1 if(NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
2   message(STATUS "Setting build type to '${default_build_type}'
3                   as none was specified")
4   set(CMAKE_BUILD_TYPE "${default_build_type}" CACHE
5       STRING "Choose type of build"
6       FORCE)
7 endif()
```

Code Block 4.11: Set Default Build Type

If the build type is not defined by the developer, it defaults to `default_build_type`, and informs the developer that it is set. Option `FORCE` is used, which means that if CMake is reconfigured without being given a build type, `CMAKE_BUILD_TYPE` will be overwritten with the default value.

Project options are also defined in this file.

```
1 option(CMAKE_DEBUG
2   "Enable various CMake debug functions, e.g. compile flag check"
3   OFF)
```

Code Block 4.12: Project Option Example

The `option` command in CMake is used to provide a boolean option to the developer that can be optionally selected when invoking CMake. This example defines the option `CMAKE_DEBUG`, gives a brief description of what the option enables or disables, and an initial value if it is not explicitly set.

In this project `CMAKE_DEBUG` is for example used in the function `add_exe_compile_flags` in DF3/cmake/checkAndApplyFlags.cmake.

4.5 Main CMakeLists

The main `CMakeLists.txt` file is in the root directory. This file is responsible for defining the project and must be present for CMake to function.

```
cmake_minimum_required(VERSION 3.27 FATAL_ERROR)

project(DF3
    VERSION 1.00
    DESCRIPTION "DF3 project in CMake"
    LANGUAGES C ASM
)
```

Code Block 4.13: CMake Project Declaration

The function `cmake_minimum_required` must be present as the first line in the main CMakeLists.txt. If it is not defined or is not called with valid arguments, CMake will throw an error and abort the configuration. It is possible to leave out the `project` function, although it is not recommended. If left out, CMake will pretend that there is a call to the function with default settings [11].

The CMakeLists.txt is responsible for including all the custom macros, functions, build options, etc. Because the main CMakeLists is in the root directory, everything included here will be globally available, unless explicitly set otherwise. This means that the CMakeLists.txt file in the subdirectory will have access to it.

```
include(buildOptions.cmake)
include(cmake/fetchFunctions.cmake)
# ... Truncated ...
fetch_df3_app_files(V${PROJECT_VERSION_MAJOR}-${PROJECT_VERSION_MINOR})
fetch_clear_module(module_1 trunk 12345) [2]
# ... Truncated ...
add_subdirectory(${CMAKE_SOURE_DIR}/src)
```

Code Block 4.14: Include Build and Option files

After the inclusion of all helper files, the source files are fetched from SVN, using the macros defined in `fetchFunctions.cmake`. When everything is fetched, a call to CMake's function `add_subdirectory` is made, adding the `DF3/src` folder to the configuration. CMake then looks in the added directory for a CMakeLists.txt file and immediately processes it before continuing in the current CMakeLists.txt file.

4.6 Source CMakeLists

In the second CMakeLists.txt located in the `DF3/src` directory, everything related to the output executable is described. This includes appending compile and link flags, adding source files,

[2] module name and revision redacted

and including interface directories to the executable.

```
add_executable(${PROJECT_NAME})

# Add compile and link flags defined in DF3/buildFlags.cmake
add_exe_compile_flags(${PROJECT_NAME} PRIVATE)
add_exe_build_type_compile_flags(${PROJECT_NAME} PRIVATE)
add_exe_linker_flags(${PROJECT_NAME} PRIVATE)

# Output Conversion, CRC and Memory Map calls
convert_to_hex(${PROJECT_NAME})
crc32(${PROJECT_NAME})
target_linker_map(${PROJECT_NAME})
```

Code Block 4.15: Add Executable, Append Flags, and Call Conversion Functions

The functions previously described for adding executable flags and conversions are called here. All must be called after the executable has been defined in the `add_executable` function. The two functions `convert_to_hex` and `crc32` are called here, but will only be executed after the project is built, due to the variable `POST_BUILD` being set within. See Code Block 4.4.

`add_executable` takes in the argument of the project, and adds an executable to be built from any source files associated with it. The file name of the executable will, when cross-compiling with `armclang`, be "`${PROJECT_NAME}.elf`". This function has been overwritten as shown in Code Block 4.2, to include the build type associated scatter file.

The rest of the file consists of specifying the source files for the executable and listing the interface directories it should search when looking for included header files.

```
target_sources(${PROJECT_NAME} PRIVATE
    ${module_1_SOURCE_DIR}/source_1.c [3]
    ${module_1_SOURCE_DIR}/source_2.c
    # ... )
# ... Truncated ...
target_include_directories(${PROJECT_NAME} PRIVATE
    ${module_1_SOURCE_DIR}/if [4]
    ${module_2_SOURCE_DIR}/if
    # ... )
```

Code Block 4.16: Adding Sources and Interface Directories

`target_sources` and `target_include_directories` have the same general syntax. The first input argument is the target for the following list of files or directories. A keyword of either `INTERFACE`, `PUBLIC`, or `PRIVATE` is given, which specifies the scope of the files or directories. For `target_sources` a list of source files used for building the target is provided. For `target_include_directories` directories are listed, and they are used when the compiler processes `#include` directives in the source code.

[3] filenames redacted
[4] modulenames redacted

The source files and include directories were first created as multiple static and interface libraries. A static library was created for each included module and all relevant source files were listed for the linking process. The header files were likewise collected into interface libraries for easier separation and readability. This method was shown to work very well until LTO (*Link Time Optimization*) flags were added to the compilation and linking process. The workaround for this issue was to target all the source files directly to the executable and have them all linked to it directly. The reason for this behavior was later discovered in the `ARM compiler 6` toolchain documentation. The toolchain linker `armlink` does not support bitcode objects from libraries [12]. In other words, source files cannot be compiled with LTO enabled, linked to libraries, before being linked to create the executable.

An option has been made to set specific compile flags for a single source file.

```
if(DEFINED SOURCE_FILE AND DEFINED COMPILE_FLAG)
  file(GLOB_RECURSE TARGET_FILE
       "${FETCHCONTENT_BASE_DIR}/${SOURCE_FILE}")
  set_source_files_properties(${TARGET_FILE} PROPERTIES
                              COMPILE_FLAGS "${COMPILE_FLAG}")
endif()
```

Code Block 4.17: Add Compile Flag to Source File

The `file` function searches all the subdirectories in `FETCHCONTENT_BASE_DIR` to find `SOURCE_FILE`. If found, the output variable `TARGET_FILE` will be set to the complete path to the file. When the source file and compile flag variables are set, CMake will apply the flag for that specific file during compiling.

For the developer to specify the source file and compile flag, CMake needs to set these values when invoked.

```
cmake -B <builddir> -G Ninja -DSOURCE_FILE="source_1.c" -DCOMPILE_FLAG="-O2"
```

Code Block 4.18: Invoke CMake with Source File and Compile Flag

4.7 MakeFile Shim

To better interact with the CMake system and create a standardized interface to apply several arguments to CMake without having to write them out every time, a Makefile has been created at the root of the directory. In the Makefile, several targets are defined to invoke CMake, Ninja, reconfiguration, etc.

```
BUILDRESULTS ?= buildresults

.PHONY: default
default: | $(BUILDRESULTS)/build.ninja
  $(Q)ninja -C $(BUILDRESULTS)

```

```
# Runs whenever the build has not been configured successfully
$(BUILDRESULTS)/build.ninja:
  $(Q)cmake -B $(BUILDRESULTS) $(OPTIONS) $(INTERNAL_OPTIONS)
```

Code Block 4.19: Makefile Targets

The `default` target has an order-only dependency (defined by "|") on the build.ninja file inside of the build results directory. The order-only dependency will be built before `default` if it does not exist, but any changes to the dependency will not cause `default` to be rebuilt.

The `OPTION` variable can be set by the developer when invoking Make, and it will be appended when CMake is invoked. `INTERNAL_OPTIONS` are a set of predefined options within the Makefile and could, for example, be an option to enable LTO, decide the build type, etc.

```
BUILD_TYPE ?=
ifneq ($(BUILD_TYPE),)
  INTERNAL_OPTIONS += -DCMAKE_BUILD_TYPE=$(BUILD_TYPE)
endif

LTO ?=
ifneq ($(LTO),)
  INTERNAL_OPTIONS += -DENABLE_LTO=ON
endif
```

Code Block 4.20: Makefile Internal Options

With the implementation of this file, a new way to invoke the system and build has been created. For example:

```
make LTO=1 BUILD_TYPE=Debug
```

Code Block 4.21: Invoking Build With Make

4.8 Summary

This chapter showed the implementation details of the project components and showed the most important parts of the code with individual descriptions. The implementation has made it possible to accommodate most of the project objectives, only leaving out a few of the "*Could have*" objectives. An implementation review and verification of the results is discussed on the next pages in chapter 5.

Chapter 5

Verification & Discussion

This chapter looks at the implementation, tests it, and verifies that it meets the project objectives. It comment on how the different parts were made possible and if any problems were encountered during implementation and verification. A presentation of the demonstrator was made to DLSW and the feedback is included and discussed.

This chapter proceeds to discuss future perspectives for setting up an automated build system using CMake, mentioning identified shortcomings of the project and possible ways to solve it. The project as a whole is discussed, looking at the planned time schedule compared to used time, and more.

5.1 Implementation Review

The implementation is set up to achieve all the objectives listed related to the demonstration software in the 1.3 sections Must Have, Should Have, and part of Could Have.

5.1.1 Must Have Implementation

Must Have (c): This objective specifies that there must be a demonstration software and that it must be able to generate a *.hex output of the executable, using the Arm Compiler 6 for embedded.

To use the Arm Compiler 6 toolchain on larger projects, a professional license is required. The project is able to build with the toolchain by using DLSW's `KEIL MDK-ARM` license, accessed via a FlexNet licensing. This license is configured within the IDE μVision Keil. Furthermore, the compiler directory must be placed in the Keil install path.

```
Keil_v5
└── ARM
    └── ARMCompiler6
        ├── bin
        └── ...
```

If the compiler license is not setup properly or the compiler is not placed within this specific folder, the license check will not complete and the project will fail when CMake tests the compiler at the start of configuration.

Once the toolchain and license are set up, the objective is tied to the implementation in section "Source CMakeLists", the toolchain files layered through "STM32G03x.cmake", and the hex conversion function in "conversions.cmake". When CMake is invoked with the toolchain file, it will build the executable with the listed source files and include directories, before converting the *.elf output to a *.hex file.

Must Have (d): The demonstration software must integrate with an SVN server to fetch the desired SW for the project. The implementation that makes this possible is included in "fetchFunctions.cmake". The macros defined here are used in the main CMakeLists.txt file "Main CMakeLists".

5.1.2 Should Have Implementation

Should Have (a): The demonstration software should use DLSW's CLEAR SW platform. The DLSW project chosen to set up as a demonstration software in CMake was specifically chosen because of the use of the CLEAR platform, and almost everything that is fetched from the SVN server is from this platform.

Should Have (b): It is easy to switch between different toolchains. To switch between different toolchains, a new toolchain file must be created. If, for example, the armcc toolchain is desired, an `armcc.cmake` file can be created, setting the same variables as in "armclang.cmake" but with armcc relevant values (setting `CMAKE_C_COMPILER` to "armcc" instead of "armclang").

However, it is not fully implemented in the demonstration project, since it requires more than just substituting the compiler to make for a successful new build. If this project is tried compiled with the armcc toolchain, the compile flags for the project are no longer valid. Furthermore, the source code syntax also needs to be compliant with the compiler used. Correcting all this would require an amount of work and time that has not been available or relevant for this project.

Should Have (c): The system should be independent of the use of a specific IDE. The demonstration project is inherently IDE independent since it can be invoked from the command line. Different IDE's might offer better syntax highlighting, auto completion, or hover text than others, but none of them are required to use the system.

5.1.3 Could Have Implementation

Could Have (a): The system could have options to set different compiler settings. This objective has been implemented on a general level by choosing between different build types. This can be done by defining the build type directly from the command line. This is described in section 4.7 and the build type specific settings are found in overrideBuildTypeSettings.cmake.

File specific compiler settings can be set by defining the variables described in Code Block 4.17

5.2 Verification

After the development of the system, a series of tests were carried out to verify its functionality. The detailed objectives found in the project goals section 1.3, have been broken up and rewritten to better suited for testing.

5.2.1 The project must compile with `Arm Compiler 6 for Embedded`

5.2.2 The project must produce a *.hex file output.

5.2.3 The project must fetch SW from an SVN server.

5.2.4 The project should include CLEAR SW.

5.2.5 The project should be able to easily switch between toolchains.

5.2.6 The project should be independent from a specific IDE.

5.2.7 The project could have options to set different compiler settings.

Furthermore, the output of the system has been tested on the target hardware (DF3).

The method of testing the system during development has been to print out various variables in the configuration stage, to check if the values within the files are as expected. The CMake function `message(DEBUG "<debug_message>"` is set and called throughout the code. These messages will only be printed to the terminal by setting `CMAKE_LOG_LEVEL=DEBUG`.

5.2.1 Compile with Arm6

To verify that the project can be compiled with the Arm Compiler 6, the toolchain file for the processor `STM32G03x` was passed to CMake when invoking the build. The expected result was that CMake identifies the C compiler as armclang 6.16, the project was built, and an *.elf output was produced.

```
make CROSS=STM32G03x
Re-run cmake no build system arguments
-- Building for: Ninja
-- The C compiler identification is ARMClang 6.16.0
-- The ASM compiler identification is ARMCC
```

```
6  -- Found assembler: C:/Keil_v5/ARM/ARMCompiler6.16/bin/armasm.exe
7  -- Detecting C compiler ABI info
8  -- Detecting C compiler ABI info - done
9  # ... Truncated ...
10 [139/139] Linking C executable src\SW03002025.elf
```

Code Block 5.1: CMake Configuration Output

As seen in Code Block 5.1 lines 4, 7, and 8, the compiler was successfully identified and the ABI (*Application Binary Interface*) detection was successful. The module fetching and compilation steps have been truncated and in line 10 the toolchain linker is linking the object files to create an executable *.elf file.

5.2.2 Produce *.hex file output

To verify the implementation function to convert from *.elf to *.hex format, the build output was investigated when invoking Ninja with verbose output and by checking if the expected *.hex file was in the output folder. In the build output, it was expected that the toolchain executable `fromelf` would be invoked with the correct values.

```
1 ninja -C buildresults --verbose
2 # ... Truncated ...
3 [139/139] ...
4    && fromelf --i32combined --output=SW03002025V1-00.hex SW03002025.elf
```

Code Block 5.2: Build Output: Fromelf Conversion to *.hex

When examining the build result folder, the expected *.hex file was found.

```
DF3
├── buildresults
│   ├── ...
│   └── src
│       ├── SW03002025.elf
│       ├── SW03002025V1-00.hex
│       └── ...
└── ...
```

5.2.3 Fetch From SVN

When a module has been successfully fetched from the repository, the variable `${name}_ADDED` is set to true. To verify that the fetch functions work, this variable was checked, and the content was manually inspected and compared with the content of the files on the SVN server. The content was fetched at different revisions, and each time the content matched the SVN repository at the given revision.

5.2.4 Include CLEAR Software

This objective closely links to the previous section and was verified when a URL was passed to the SVN CLEAR directory and the fetching succeeded.

5.2.5 Switch Toolchains

To verify that switching between toolchains was possible and could be easily done, a new toolchain `armcc.cmake` was created and included instead of the `armclang.cmake` toolchain in the `cortex-m0plus.cmake` toolchain file.

```
- include(${CMAKE_CURRENT_LIST_DIR}/armclang.cmake)
+ include(${CMAKE_CURRENT_LIST_DIR}/armcc.cmake)
```

`armclang.cmake` has been copied and rewritten to set `CMAKE_C_COMPILER` to `armcc` instead of `armclang`

```
- set(CMAKE_C_COMPILER armclang)
+ set(CMAKE_C_COMPILER C:/Keil_v5/ARM/ARM_Compiler_5.06u5/bin/armcc.exe)
```

When invoking Make using the expected output is that CMake identifies the compiler to be armcc. As described in the implementation section 5.1.2 Should Have (b), the project is not expected to build with this configuration.

```
make CROSS=STM32G03x
Re-run cmake no build system arguments
-- Building for: Ninja
-- The C compiler identification is ARMCC 5.6.528
```

Code Block 5.3: CMake Configuration Output, armcc

CMake was able to identify the compiler when switching from one toolchain file to another.

5.2.6 IDE Independency

CMake is inherently IDE independent and only needs a text editor tool for the development of the system itself. Using specific editors, like VS Code, does offer help in the form of syntax highlighting, hover information, and auto-completion, but is in no way a requirement. To invoke CMake, a command-line interface like PowerShell or Command Prompt is required.

5.2.7 Set Compiler Settings

To test the project's ability to set different compiler settings, a series of different builds were carried out, changing the build type and setting flags for individual source files. To verify the output, the `compile_commands.json` file in the build directory was investigated.

```
DF3
├── buildresults
│   ├── ...
│   └── compile_commands.json
└── ...
```

For the Release build, it is expected to find the Release exclusive flags `"-flto"` and `"-DSOFTWARE_NUMBER=1"`.

```
{
  "directory": "C:/thesis/df3/buildresults",
  "command": "C:\\...\\armclang.exe ... -flto -DSOFTWARE_NUMBER=1
              -o src\\...\\source_1.o -c C:\\...\\source_1.c",
  "file": "C:\\...\\source_1.c",
  "output": "src\\...\\source_1.o"
}, [1]
# ...
```

Code Block 5.4: DF3 Release Build

For the Debug build, it is expected to find the Debug exclusive flags `"-DDEBUG"` and `"-DSOFTWARE_NUMBER=2"`.

```
{
  "directory": "C:/thesis/df3/buildresults",
  "command": "C:\\...\\armclang.exe ... -DDEBUG -DSOFTWARE_NUMBER=2
              -o src\\...\\source_1.o -c C:\\...\\source_1.c",
  "file": "C:\\...\\source_1.c",
  "output": "src\\...\\source_1.o"
}, [1]
# ...
```

Code Block 5.5: DF3 Debug Build

A new compile optimization flag was passed to one of the source files, which was also reflected in the output

```
make BUILD_TYPE=Debug SOURCE_FILE="source_1.c" COMPILE_FLAG="-O1"
```

```
{
  "directory": "C:/thesis/df3/buildresults",
  "command": "C:\\...\\armclang.exe -DSOFTWARE_NUMBER=2 -O1
              -o src\\...\\source_1.o -c C:\\...\\source_1.c",
  "file": "C:\\...\\source_1.c",
  "output": "src\\...\\source_1.o"
}, [2]
```

Code Block 5.6: DF3 Setting Compile Flag For ”source_1.c”

[1] Output truncated (...) and filenames redacted
[2] Output truncated (...) and filenames redacted

5.2.8 Test On Target

To verify that the output *.hex file was valid, it was programmed on the DF3 hardware using LINAK's *LIN Programmer* tool. The expected outcome of the test was that it could be programmed without warnings and, when connected to a control handset, it would be able to start an initialization sequence.

The test was carried out together with two DLSW architects and the outcome was as expected, without any warnings or other complications.

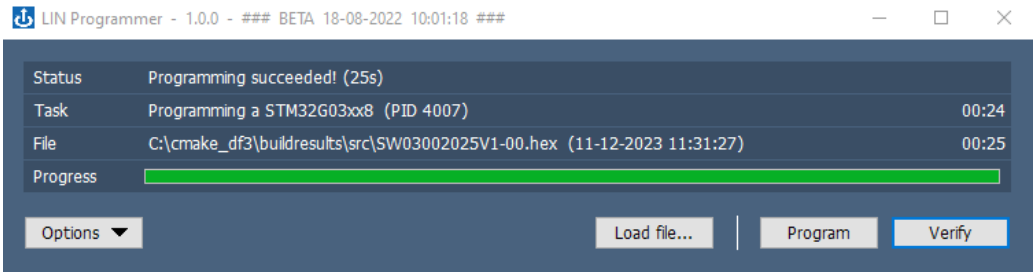


Figure 5.1: Successful DF3 programming

5.2.9 Performance Comparison

A performance comparison between CMake/Meson and Keil was not carried out in req#2.7.1 analysis, as explained. During the implementation, it was observed that the build time when using CMake and Ninja was visibly faster than when building with Keil. This was recorded, and measurements were made for both systems. Both systems were set to build with parallel jobs equal to the amount available on the build machine. Ten sets of measurements were taken for each system and an average time was calculated.

Build System	Avg. time [s]	Min. time [s]	Max. time [s]
CMake (Ninja)	23.1	20.4	23.9
Keil	41.3	40	43

Table 5.1: Perfomance Comparison Cmake and Keil

The results show that building with CMake and Ninja is approximately 79% faster than the build time in Keil. This is a significant performance improvement and will save developers a lot of time when developing projects in the future. Measurements can be seen in Appendix section D.6.

5.3 Presentation at LINAK

The end of the project included a presentation of the demonstration project to select members of DLSW and two developers / architects from LINAK's MEDLINE & CARELINE software

department. The presentation included a walk-through of the project structure and the implemented functionality. It provided the teams with preliminary information on what is possible when building a project with the help of CMake.

5.3.1 Presentation Structure

The presentation started with a quick explanation of the project directory and file structure. This was followed by a detailed walk-through of the following:

- The three toolchain files
- User defined functions
- Build options and flag definitions
- The projects CMakeLists
- Makefile interface

After the project had been presented, a short reflection on key requirements where mentioned, including *Reproducibility*, *Performance*, *Ease of Use*, and *IDE Compatibility*.

Identified current shortcomings of the project where mentioned, including the inability to fetch directories from the SVN server that has either been renamed or has been deleted entirely. For the demonstration software, workarounds were made by fetching renamed content under its new name but at the same "old" revision. No deleted file or directory was encountered during implementation, and a solution has not yet been investigated further.

5.3.2 Presentation Feedback

The participants were generally content with the demonstration and excited to see how this can be optimized and scaled in the future. The presentation showed that it is possible to create a *Keil* equivalent build using CMake. It showed that the chosen DF3 project could be built with Arm Compiler 6 and run on target hardware. Furthermore, it was shown that the CMake build is better than *Keil* in areas such as build time performance.

The participants agreed that a general structure, a build-framework, must be created when scaling the project up in the future. Several things such as available parameters, build configurations, structure of cmake files and their individual responsibilities, and the local directory structure. All common and project-specific items must be identified and clearly separated. This way all common items can be collected and only exist in one place, avoiding any redundancy in work when creating new projects.

Participants discussed the importance of being able to fetch deleted or renamed content. They came to the conclusion that the ability to achieve this is a requirement that is of the utmost importance. Files and directories will be merged, renamed, or deleted during the evolution of the CLEAR platform, and this should not have an impact on the ability to build older projects.

5.4 Future Perspectives

This section will list future possibilities and recommendations when scaling the build system to encompass more than a single project.

5.4.1 Individual Module CMakelists

DLSW already has a CMake structure in place, where all CLEAR modules have their own CMakeLists.txt file in the root directory. These files create libraries and interfaces for the individual modules to be included and used in other CMakeLists. However, these lists cannot be used in a project setup in their current state. The main reason for this is that the armclang compiler with LTO enabled cannot link static libraries. This was described in section 4.6.

The current CMakeLists needs to be restructured, and one way to do this is to create lists of source files and pass the source files directly to the project CMakeLists for further processing.

By doing this, the project CMakeLists does not need to have knowledge about the internal structure of an included module. This could help reduce the maintenance of the project, as it does not need to have white-box knowledge of all included modules.

5.4.2 Debugging Facilities

Another important factor for future development when scaling the project is the use of debug facilities. Debugging is a large part of developing embedded software and a solution must be found to solve this. The current method is to debug within *Keil*, which uses ARM's CMSIS *(Common Microcontroller Software Interface Standard)*. There exist extensions for VS Code to debug arm processors using the CMSIS library, and this would be a good place to start when investigating debug methods in the future.

5.4.3 Fetching of Renamed or Deleted Modules

A feature that must be investigated and found a solution for, is the ability to fetch content from the SVN server that has been renamed or deleted. The importance of which is mentioned in 5.3.2 Presentation Feedback. There might be an SVN-specific solution or a solution within CMake to achieve this. On the basis of the knowledge of CMake and the widespread usage in the industry for a number of years, it is expected that a solution to this problem can be found.

5.4.4 Integration With Jenkins

This was part of the "Could Have" objectives for the project and was not implemented during development. This makes it a great candidate to investigate a solution for when scaling the project. DLSW already has a structure to run CMakelists through Jenkins with their unit testing structure. To implement this functionality for the project, a `Jenkinsfile` must be created at the root of the project.

5.5 Course of Events

The project progressed as expected and only a few obstacles were encountered along the way. The risks and mitigation strategies identified at the start of the project provided great support throughout the project. As the possible risk described in Risk Table A.5, implementation and / or compatibility issues were identified when a new tool or new system functionality was used. These were mitigated by creating a smaller and simpler implementation to test the issue at the core in order to find a proper solution. The best example of this was found when implementing the executable details in section 4.6.

5.5.1 Time Plan Evaluation

As seen in Appendix A: Problem Formulation, an initial time plan was made for the project. The plan was followed only with minor changes that affected the date that milestone #2 was met. Milestone #2 defined the transition from research and analysis to design and implementation, with the choice of a build system for the demonstration setup. Originally it was planned to be reached on the 31st of October, before the 2 week personal time off, but ended being decided on the meeting held on Tuesday the 21st of November. It is a push of 21 days, but when subtracting the time away, it only set back the project by three days. The time lost was reclaimed during the development of the demonstration software, which was presented to DLSW on Monday the 11th of December. The final time schedule can be found in Appendix A.2

Chapter 6

Conclusion

This project created a demonstrator software showing how it is possible to set up a software project using the build system generator CMake. It did so by first investigating, analyzing, and comparing the two build system generators, CMake and Meson. It considered a list of requirements agreed upon with DLSW for the analysis. On the basis of the analysis, DLSW decided to abandon Meson and develop the demonstrator software using CMake. The demonstrator was designed, implemented, and its functionality was verified before a list of future scaling suggestions and possibilities were given.

The demonstrator software using CMake shows that it is possible to set up an already existing project from DLSW, and have it running on the target hardware.

The project meets all the objectives listed in section 1.3 *Must Have*, *Should Have*, and parts of *Could Have*. It did not provide a method to debug a *.hex file on the hardware target, and the demonstrator software is not running on a Jenkins server.

Bibliography

In cases where URLs are broken and spread on several lines, press the last line of the URL.

[1] LINAK A/S. *About LINAK*. https://www.linak.com/about-linak/. (Accesed on 12/17/2023).

[2] LINAK A/S. *Innovation*. https://www.linak.com/about-linak/innovation/. (Accesed on 11/09/2023).

[3] Kitware + CMake Community. *CMake Wiki*. https://gitlab.kitware.com/cmake/community/-/wikis/home. (Accesed on 10/24/2023).

[4] Kitware + CMake Community. *Projects*. https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/Projects. (Accesed on 10/24/2023).

[5] Meson + Community. *meson discussion*. https://github.com/mesonbuild/meson/discussions. (Accesed on 10/25/2023).

[6] Github. *meson - contributers*. https://github.com/mesonbuild/meson/graphs/contributors. (Accesed on 10/26/2023).

[7] Kitware. *CMake*. https://cmake.org/. (Accesed on 10/24/2023).

[8] Kitware. *CMake Cache*. https://cmake.org/cmake/help/book/mastering-cmake/chapter/CMake%20Cache.html. (Accesed on 12/05/2023).

[9] Kitware. *cmake-toolchains(7)*. https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html. (Accesed on 10/18/2023).

[10] Kitware. *Licensing*. https://cmake.org/licensing/. (Accesed on 10/20/2023).

[11] Kitware. *project*. https://cmake.org/cmake/help/latest/command/project.html. (Accesed on 12/05/1995).

[12] Arm Limited. *Arm Compiler User Guide Versoin 6.16: Restrictions with Link-Time Optimization*. https://developer.arm.com/documentation/100748/0616/Writing-Optimized-Code/Optimizing-across-modules-with-Link-Time-Optimization/Restrictions-with-Link-Time-Optimization?lang=en. (Accesed on 12/13/2023).

[13] Evan Martin. *Ninja, a new build system*. https://neugierig.org/software/chromium/notes/2011/02/ninja.html. (Accesed on 11/06/2023).

[14] Robert Maynard. *CMake 3.0.0 Released*. https://cmake.org/pipermail/cmake/2014-June/057793.html. (Accessed on 09/20/2023).

[15] Lars Melchior. *Setup-free CMake dependency management*. https://github.com/cpm-cmake/CPM.cmake. (Accesed on 12/03/2023).

[16] Meson. *Built-in options*. https://mesonbuild.com/Builtin-options.html. (Accesed on 10/17/2023).

[17] Meson. *Legal information*. https://mesonbuild.com/legal.html. (Accesed on 10/20/2023).

[18] Meson. *List of projects using Meson*. https://mesonbuild.com/Users.html. (Accesed on 10/24/2023).

[19] Meson. *The Absolute Beginner's Guide to Installing and Using Meson*. https://mesonbuild.com/SimpleStart.html. (Accesed on 10/24/2023).

[20] Meson. *The Meson Build System*. https://mesonbuild.com/. (Accesed on 10/17/2023).

[21] mesonbuild. *Meson*. https://marketplace.visualstudio.com/items?itemName=mesonbuild.mesonbuild. (Accesed on 10/27/2023).

[22] Microsoft. *CMake Tools*. https://marketplace.visualstudio.com/items?itemName=ms-vscode.cmake-tools. (Accesed on 10/27/2023).

[23] Eric Steven Raymond. *The Art of Unix Programming*. 1st ed. Addison-Wesley, 2023. ISBN: 978-0-131-42901-7.

[24] Gigi Sayfan. *A Build System For Complex Projects: Part 1*. https://web.archive.org/web/20161105185620/http://www.drdobbs.com/tools/a-build-system-for-complex-projects-part/218400678. (Accessed on 10/04/2023).

[25] Tim Schürmann. *Meson - a new build system*. https://www.linux-magazine.com/Issues/2014/166/Meson-Build-System. (Accesed on 10/26/2023).

[26] Craig Scott. *Proffesional CMake: A Practical Guide*. 15th ed. Craig Scott, 2023. ISBN: 978-1-925904-24-6.

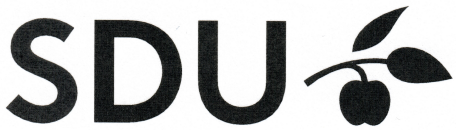
[27] Stackshare. *CMake*. https://stackshare.io/cmake. (Accesed on 10/24/2023).

[28] Jose Torres. *CMake Language Support*. https://marketplace.visualstudio.com/items?itemName=josetr.cmake-language-support-vscode. (Accesed on 10/27/2023).

[29] Martin Weber. *CMake Editor*. https://marketplace.eclipse.org/content/cmake-editor. (Accesed on 10/27/2023).

Appendix A

Problem Formulation



Automated Embedded Project Setup

Problem Formulation

Søren K. Haug

B.Eng. Electronics

SDU Supervisor: Associate Professor Søren Top

Fall semester 2023

External Company: LINAK A/S

LINAK Supervisor:

Anders Kristensen, Platform Architect, DESKLINE

Assoc. Prof. S. Top, SDU	A. Kristensen, LINAK	S.K. Haug
27/9 2023	27/9-23	27/9-23
Date	Date	Date

1 Company Background

LINAK was established by the current CEO and owner Bent Jensen's grandfather in 1907. After Bent Jensen took over the company from his father in 1976, he revolutionized it by inventing the electric linear actuator in 1979. With the introduction of this technology, LINAK quickly established itself as a leader in the industry.

Today, LINAK is still an industry leader with a commitment to innovation and a commitment to providing the highest-quality products and services to customers around the world. As of 2019 LINAK counted more than 2400 employees spread across 35 countries. The headquarters is located in the town of Guderup in southern Denmark covering an area of 55,000m^2. 1250 employees work in the Guderup headquarter within manufacturing, group management, R&D, logistics, and administrative functions. LINAK has established factories outside of Denmark in USA, China, Slovakia, and Thailand.

LINAK consists of 4 business segments.

- DESKLINE specialises in ergonomic systems for office desks, workstations, kitchens, and shop and conference room interiors
- HOMELINE excels within comfort furniture
- TECHLINE handles durable actuator solutions for heavy-duty applications operating in the harshest of environments
- MEDLINE & CARELINE focuses solely on developing intelligent systems for healthcare applications

2 Project Goals

LINAK's DESKLINE segment is experiencing an increasing challenge in the development process for their software. With a growing software platform, the time it takes developers to setup and build projects is increasing. Setting up projects manually can lead to the introduction of human errors, which can further increase the amount of resources spent during this process. DLSW (*DESKLINE Software*) is currently limited to using the IDE μVision Keil to set up, build, and debug their software. The use of μVision Keil also has a considerable annual cost associated with it.

One way to improve the work process of individual developers and reduce the cost associated with building a software project is by implementing a build automation tool. Automating the build process can help reduce the time an individual developer uses to set up projects, almost eliminate human errors, and improve overall development efficiency. Therefore, the goal is to investigate the available build automation tools and propose the best solution to achieve automation when building projects.

2.1 Objectives

In order to achieve the goal, several objectives can be identified. These will each assist in proposing a build automation tool.

- Research automated software building procedures to gain knowledge and learn from previous experiences.
- Investigation of two different build automation tools to get a deeper understanding of which features they offer.
- An analysis of the automation tools and compare them against each other according to their features and the needs of DLSW. Select the most relevant to use.
- How does the solution(s) fit in with DLSW's current CI strategy?
- Design and create a prototype software using the chosen tool.

2.2 Main Activities

2.2.1 Investigate Build Automation

A build automation system in an embedded environment automates software compilation, dependency management, and binary generation. In this part, the general theory behind build automation is investigated along with any related languages.
This will include investigation of the Cmake software, the build automation tool Make and relevant theory behind. A second automation tool is identified for further investigation.

2.2.2 Investigate Automation Tools

Two different automation tools are further investigated, one of which is Cmake. They will be investigated with focus on their individual features, compatibility with DLSW's current practices and other relevant factors.
All identified criteria will serve as input to the analysis and comparison of the tools.

2.2.3 Analysis of Automation Tools

An individual analysis of the Cmake build automation software and the second chosen software. Analyze how implementing a build automation software will improve the current CI strategy at DLSW. The analysis is based on the criteria identified in the investigation above.
The two methods will be compared to each other based on the analysis and will be presented to the DLSW manager and architects for strategic decision making.

2.2.4 Prototype Software Implementation and Verification

The prototype software will be made using the chosen automation software. This process involves showcasing essential key features, testing the usability, etc. The implementation will be tested using a combination of different modules both locally and with the integration of fetching modules from an SVN server. This will be used to determine the feasibility of a full-scale implementation at DLSW.

This part will also describe any problems encountered during the testing process and will be handed over to DLSW together with the theory and prototype software.

2.3 Detailed objectives

1. **Must Have**
   (a) Investigation of build automation and Makefiles.
   (b) Analysis of two different build automation solutions.
      - CMake must be one of these.
   (c) A prototype software implementation:
      - Must generate a project .hex file using ARMv6 compiler.
   (d) Integration with an SVN server to fetch desired SW.
2. **Should Have**
   (a) Include project setup using DLSW's CLEAR SW platform.
   (b) Ability to easily switch between different compilers.
      - ARMv5, ARMv6, gcc
   (c) Be independent from the use of a specific IDE.
3. **Could Have**
   (a) Options to set different compiler settings (optimization etc.).
   (b) Investigation and analysis of additional solution(s).
   (c) Have the prototype software running on a Jenkins server.
   (d) A method to debug a .hex file, built using the chosen solution, on a hardware target.
4. **Won't Have**
   (a) Continuous delivery/deployment analysis.
   (b) Production SW solution for DLSW.

2.4 Limitations

Due to the amount and complexity of available tools for automating embedded builds, this project will be limited to the examination of two different tools.
The following different tools are selected for investigation

- Cmake
- Meson or Bazel

Due to the scale of implementing a build automation tool into the already established work environment at DLSW, the project will not provide any production software. It will investigate the feasibility of using an automation tool and showcase essential functionality.

3 Work Methodology

The approach for the project will be similar to that of the V-model. To start the project, all relevant theories of automation software will be investigated. Based on this investigation two solutions will be subject for further investigation. This will lead to an analysis of the two solutions, comparing them to each other, and presented to DLSW for the strategic choice of which one to start developing. Then a prototype software showcasing the chosen solution will be created, followed by testing and evaluation.

Regular meetings will be held with the company and university supervisors respectively.

3.1 Risk Assessment

Before the project starts, all relevant risks have been identified. Each risk has been rated according to probability and impact on the project. For each of the risks a mitigation strategy has been identified.

Risk tables can be found in the appendix.

3.2 Initial Time Schedule

The initial time schedule shows the main activities of the project, any subcategories to these, and allocates time to each phase. It is initially divided into 3 main phases, **Research and Investigation**, **Analysis** and **Prototype SW**. These reflect the main activities and roughly the different parts of the project report.

There are 3 milestones in the project, marked with ♦ in the schedule.

1. 11^{th} of October: Criteria for automation system agreed upon by all stakeholders
2. 30^{th} of October: Choice of automation system for prototype development
3. 6^{th} of December: Presentation of prototype solution

Time schedule on the next page.

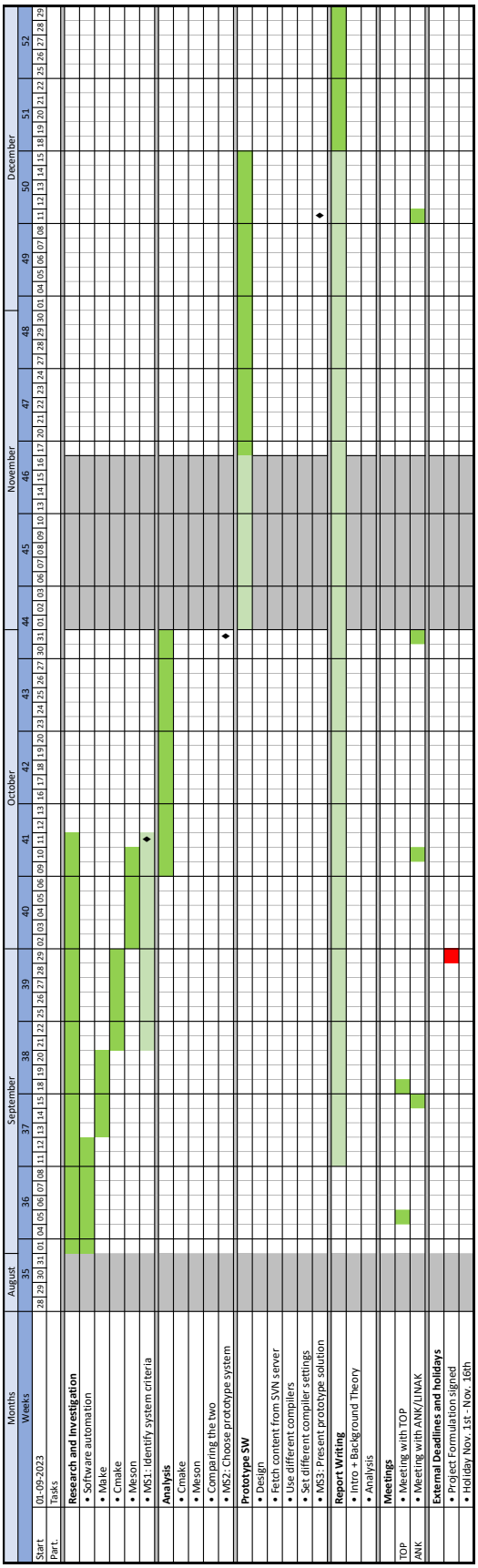


Figure 1: Time schedule version 1.0

A.1 Identified Risks

Risk: Unavailability of LINAK stakeholders for feedback and decision making.	
Probability	Low
Severity	Medium
Mitigation	Regularly work at the LINAK office to maintain open communication and ensure timely feedback and decision making.

Table A.1: Risk: Unavailability of LINAK stakeholders for feedback and decision making.

Risk: The learning curve of the individual tools is too steep, leading to an excessive amount of time spent on learning.	
Probability	Medium
Severity	Medium
Mitigation	Build investigation prototypes during research to get a grasp of the time needed as soon as possible.

Table A.2: Risk: The learning curve of the individual tools is too steep, leading to an excessive amount of time spent on learning.

Risk: LINAK work laptop breaks down, leading to loss of data and access to LINAK SVN server.	
Probability	Very Low
Severity	High
Mitigation	Back up report, notes, code, etc. outside of the work laptop hard drive. Continue working on other tasks until another laptop is ready.

Table A.3: Risk: LINAK work laptop breaks down, leading to loss of data and access to LINAK SVN server.

Risk: Compatibility issues between investigated tools and LINAK's workflow, codebase etc.	
Probability	Low
Severity	High
Mitigation	Involve stakeholders at LINAK in the decision process and test tools with DLSW's codebase

Table A.4: Risk: Compatibility issues between investigated tools and LINAK's workflow, codebase etc.

Risk: Compatibility issues between chosen tools, libraries and development environments.	
Probability	Low
Severity	Medium
Mitigation	Test compatibility early and ensure all tools work seamlessly together.

Table A.5: Risk: Compatibility issues between chosen tools, libraries and development environments.

Risk: Lack of documentation of the chosen tools, LINAK's procedures and their code base.	
Probability	Low
Severity	Medium
Mitigation	Investigate available material early in the project to ensure available documentation is gathered in a timely manner

Table A.6: Risk: Lack of documentation of the chosen tools, LINAK's procedures and their code base.

A.2 Final Time Schedule

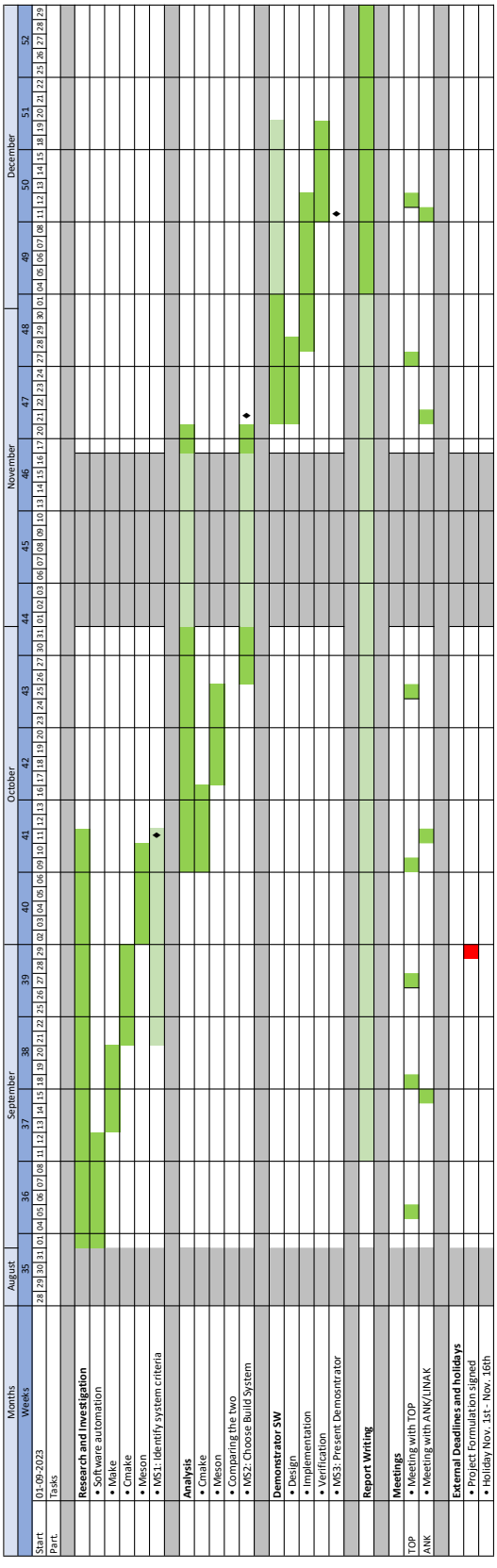


Figure A.1: Final Time Schedule

Appendix B

Test Project

B.1 Source Files

cmake\lib\v.h

```h
#ifndef V_H
#define V_H

#define YCONST 42

int subtract_new(int a, int b);
double multiply_new(double a, double b);
double divide_new(double a, double b);
int square_new(int x);
int cube_new(int x);

#endif  /* V_H */
```

cmake\lib\x.h

```h
#ifndef X_H
#define X_H

#define YCONST 42

int subtract(int a, int b);
double multiply(double a, double b);
double divide(double a, double b);
int square(int x);
int cube(int x);

#endif  /* X_H */
```

cmake\lib\z.h

```h
#ifndef Z_H
#define Z_H

#define YCONST 42

int add(int a, int b);
int repeatNtimes(int n);

#endif  /* Z_H */
```

cmake\src\a.c

```c
#include "x.h"

int subtract(int a, int b) {
    return a - b;
}
```

cmake\src\b.c

```c
#include "x.h"

double multiply(double a, double b) {
    return a * b;
}

double divide(double a, double b) {
    if (b != 0) {
        return a / b;
    } else {
        // Handle division by zero error
        return 0.0;
    }
}
```

cmake\src\c.c

```c
#include "x.h"

int square(int x) {
    return x * x;
}

int cube(int x) {
    return x * x * x;
}
```

cmake\src\d.c

```c
#include "z.h"

int add(int a, int b)
{
    return a + b;
}

```

cmake\src\e.c

```c
#include "z.h"

int b = 0;
```

```
 4
 5  int repeatNtimes(int n)
 6  {
 7      for (int i = 0; i < n; i++)
 8      {
 9          b++;
10      }
11      return b;
12  }
```

cmake\src\f.c

```
1  #include "v.h"
2
3  int subtract_new(int a, int b) {
4      return (a - b) + 2;
5  }
```

cmake\src\g.c

```
1  #include "v.h"
2
3  double multiply_new(double a, double b) {
4      return (a * b) + 2;
5  }
6
7
```

cmake\src\h.c

```
1  #include "v.h"
2
3  int square_new(int x) {
4      return (x * x) + 2;
5  }
```

cmake\src\i.c

```
1  #include "v.h"
2
3  int cube_new(int x) {
4      return (x * x * x) + 2;
5  }
```

cmake\src\j.c

```
1  #include "v.h"
2
3  double divide_new(double a, double b) {
```

B.2 CMake Files

cmake\CMakeLists.txt

```
cmake_minimum_required(VERSION 3.17)
project(CMake_quality_test
  VERSION 1.0
  DESCRIPTION "A project"
  LANGUAGES C
)

# Set build tyoe to Debug (-g)
set(CMAKE_BUILD_TYPE Debug)

# Include override function of "add_executable", to create link script dependency
include(cmake/AddExecutableWithLinkerScriptDep.cmake)
# Include custom convertion functions
include(cmake/Conversions.cmake)
# Run CMakeLists.txt in src/
add_subdirectory(src)

# Add exeuctable and specify sources and lib links.
add_executable(sample_app)
target_sources(sample_app PRIVATE ${CMAKE_SOURCE_DIR}/src/main.c)
target_link_libraries(sample_app PRIVATE
  c_lib
  new_c_lib
)

# Custom functions to create memory map
# target_linker_map(sample_app)

# Custom function to convert from ELF to .hex and .bin
convert_to_hex(sample_app)
convert_to_bin(sample_app)

```

cmake\cmake\AddExecutableWithLinkerScriptDep.cmake

```
# Redefine add_executable so that we specify a link-time dependency on the linker script
# The `LINKER_SCRIPT_DEPENDENCIES` variable should be defined in a toolchain
# file. It can contain a CMake list of all linker script dependencies.
function(add_executable target)
  # Forward all arguments to the CMake add_executable
  _add_executable(${target} ${ARGN})
  if(LINKER_SCRIPT_DEPENDENCIES)
    set_target_properties(${target} PROPERTIES
      LINK_DEPENDS "${LINKER_SCRIPT_DEPENDENCIES}")
```

```
  endif()
endfunction()

```

cmake\cmake\Conversions.cmake

```
# Custom command line function to convert from ELF to hex
function(convert_to_hex TARGET)
    add_custom_command(
      TARGET ${TARGET} POST_BUILD
      COMMAND ${CMAKE_OBJCOPY} -O ihex ${TARGET} ${TARGET}.hex
      BYPRODUCTS ${TARGET}.hex
    )
endfunction()

# Custom command line function to convert from ELF to bin
function(convert_to_bin TARGET)
    add_custom_command(
      TARGET ${TARGET} POST_BUILD
      COMMAND ${CMAKE_OBJCOPY} -O binary ${TARGET} ${TARGET}.bin
      BYPRODUCTS ${TARGET}.bin
    )
endfunction()

# Custom function to set linker flags for creating a memory map
function(target_linker_map target)
  # Hardcoded map linker flag for gcc
  set(GEN_MAP_FILE "-Wl,-Map,")
    get_target_property(map_dir ${target} BINARY_DIR)
    target_link_options(${target} PRIVATE ${GEN_MAP_FILE}${map_dir}/${target}.map)
endfunction()
```

cmake\cmake\toolchains\cross\STM32F103VBIx.cmake

```
##############################
# STM32F103VBIx (Cortex-M3) #
##############################

# CMake includes the toolchain file multiple times when configuring the build,
# which causes errors for some flags (e.g., --specs=nano.specs).
# We prevent this with an include guard.
if(STM32F103VBIx_TOOLCHAIN_INCLUDED)
  return()
endif()

set(STM32F103VBIx_TOOLCHAIN_INCLUDED true)

set(CPU_NAME STM32F103VBIx)

set(LINKER_SCRIPT_DIR ${CMAKE_SOURCE_DIR}/cmake/linker-scripts/stm)
set(LINKER_SCRIPT_PRIMARY_FILE STM32F103VBIx_FLASH.ld)

# These dependencies are applied to add_executable targets by the
```

```
20 # AddExecutableWithLinkerScriptDep module
21 list(APPEND LINKER_SCRIPT_DEPENDENCIES "${LINKER_SCRIPT_DIR}/${LINKER_SCRIPT_PRIMARY_FILE}"
22
23 set(LD_FLAGS "-T${LINKER_SCRIPT_PRIMARY_FILE} -L${LINKER_SCRIPT_DIR}")
24
25 include(${CMAKE_CURRENT_LIST_DIR}/cortex-m3.cmake)
26
```

cmake\cmake\toolchains\cross\arm-none-eabi-gcc.cmake

```
1  ####################################
2  # arm-none-eabi-gcc Base Toolchain #
3  ####################################
4  #
5  # To include this file as a base toolchain file,
6  # include it at the bottom of the derived toolchain file.
7  #
8  # You can define CPU_FLAGS that will be passed to CMAKE_*_FLAGS to select the CPU
9  # (and any other necessary CPU-specific flags)
10 # You can define VFP_FLAGS to select the desired floating-point configuration
11 # You can define LD_FLAGS to control linker flags for your target
12
13 #################
14 # System Config #
15 #################
16
17 set(CMAKE_SYSTEM_NAME Generic)
18 set(CMAKE_SYSTEM_PROCESSOR arm)
19 # Represents the name of the specific processor type, e.g. Cortex-M4
20 if(NOT CPU_NAME)
21   set(CPU_NAME generic)
22 endif()
23
24 ####################
25 # Toolchain Config #
26 ####################
27
28 set(CMAKE_C_COMPILER    arm-none-eabi-gcc.exe)
29 set(CMAKE_CXX_COMPILER  arm-none-eabi-g++.exe)
30 set(AS                  arm-none-eabi-as.exe)
31 set(CMAKE_AR            arm-none-eabi-gcc-ar.exe)
32 set(OBJCOPY             arm-none-eabi-objcopy.exe)
33 set(OBJDUMP             arm-none-eabi-objdump.exe)
34 set(SIZE                arm-none-eabi-size.exe)
35
36 # If set to ONLY, then only the roots in CMAKE_FIND_ROOT_PATH (i.e., the host machine)
37 # will be searched. If set to NEVER, then the roots in CMAKE_FIND_ROOT_PATH will
38 # be ignored and only the build machine root will be used.
39 set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
40 set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
41 set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
42 set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
43
44 # Test compiles will use static libraries, so we won't need to define linker flags
45 # and scripts for linking to succeed
```

```
46 set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)
47
48 ################
49 # Common Flags #
50 ################
51 # Note that CPU_FLAGS, LD_FLAGS, and VFP_FLAGS are set by other Toolchain files
52 # that include this file.
53 #
54 # See the CMake Manual for CMAKE_<LANG>_FLAGS_INIT:
55 # https://cmake.org/cmake/help/latest/variable/CMAKE_LANG_FLAGS_INIT.html
56
57 set(CMAKE_C_FLAGS_INIT
58   "${CPU_FLAGS} ${VFP_FLAGS}"
59   CACHE
60   INTERNAL "Default C compiler flags.")
61 set(CMAKE_CXX_FLAGS_INIT
62   "${CPU_FLAGS} ${VFP_FLAGS} "
63   CACHE
64   INTERNAL "Default C++ compiler flags.")
65 set(CMAKE_ASM_FLAGS_INIT
66   "${CPU_FLAGS} "
67   CACHE
68   INTERNAL "Default ASM compiler flags.")
69 set(CMAKE_EXE_LINKER_FLAGS_INIT
70   "${LD_FLAGS} -Wl,--no-gc-sections"
71   CACHE
72   INTERNAL "Default linker flags.")
73
```

cmake\cmake\toolchains\cross\cortex-m3.cmake

```
1 ##############
2 # Cortex-M3 #
3 ##############
4
5 # CMake includes the toolchain file multiple times when configuring the build,
6 # which causes errors for some flags (e.g., --specs=nano.specs).
7 # We prevent this with an include guard.
8 if(ARM_CORTEX_M3_TOOLCHAIN_INCLUDED)
9   return()
10 endif()
11
12 set(ARM_CORTEX_M3_TOOLCHAIN_INCLUDED true)
13
14 if(NOT CPU_NAME)
15   set(CPU_NAME cortex-m3)
16 endif()
17
18 set(CPU_FLAGS "-mcpu=cortex-m3 -mabi=aapcs -mthumb ${CPU_FLAGS}")
19 set(VFP_FLAGS "-mfloat-abi=soft ${VFP_FLAGS}")
20
21 include(${CMAKE_CURRENT_LIST_DIR}/arm-none-eabi-gcc.cmake)
22
```

cmake\src\CMakeLists.txt

```
# Add library and sources
add_library(c_lib STATIC)
target_sources(c_lib PRIVATE
    a.c
    b.c
    c.c
    d.c
  e.c
)

# Add link option -nostartfiles to not include standard startup files
target_link_options(c_lib PUBLIC "-nostartfiles")
# specify include path for header files
target_include_directories(c_lib PUBLIC ${PROJECT_SOURCE_DIR}/lib)

# Add library and sources
add_library(new_c_lib STATIC)
target_sources(new_c_lib PRIVATE
    f.c
    g.c
    h.c
    i.c
    j.c
)

# Add link option -nostartfiles to not include standard startup files
target_link_options(new_c_lib PUBLIC "-nostartfiles")
# specify include path for header files
target_include_directories(new_c_lib PUBLIC ${PROJECT_SOURCE_DIR}/lib)
```

B.3 CMake Performance Files

cmake_build_output\Perfomance_output\desktop_makefile_parallel_time_output.txt

```
Makefile (Parallel 16) build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains CMake configuration, generation and Make build time.

#######################################
# Measurement command in powershell: #
#######################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of CMake configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & cmake -B buildresults -G "Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE="cmake\toolchains\cross\STM32F103VBIx.
>> & cmake --build buildresults --parallel 16 }
# Output measured time in ms to a text file
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 2.448 s
Run 2 : 2.407 s
Run 3 : 2.407 s
Run 4 : 2.464 s
Run 5 : 2.400 s
Run 6 : 2.402 s
Run 7 : 2.400 s
Run 8 : 2.435 s
Run 9 : 2.425 s
Run 10: 2.391 s

Avg.  : 2.418 s
```

cmake_build_output\Perfomance_output\desktop_makefile_time_output.txt

```
Makefile build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains CMake configuration, generation and Make build time.

#######################################
# Measurement command in powershell: #
#######################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of cmake configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & cmake -B buildresults -G "Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE="cmake\toolchains\cross\STM32F103VBIx.
>> & cmake --build buildresults }
# Output measured time in ms to a text file
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################
```

```
Run 1 : 3.853 s
Run 2 : 3.895 s
Run 3 : 3.846 s
Run 4 : 3.830 s
Run 5 : 3.862 s
Run 6 : 3.850 s
Run 7 : 3.902 s
Run 8 : 3.824 s
Run 9 : 3.890 s
Run 10: 3.838 s

Avg.  : 3.859 s
```

cmake_build_output\Perfomance_output\desktop_ninja_parallel_time_output.txt

```
Ninja build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains CMake configuration, generation and Ninja build time

#######################################
# Measurement command in powershell: #
#######################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of cmake configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & cmake -B buildresults -G Ninja -DCMAKE_TOOLCHAIN_FILE="cmake\toolchains\cross\STM32F103VBIx.cmake"
>> & cmake --build buildresults }
# Output measured time in ms to a text file
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 1.555 s
Run 2 : 1.590 s
Run 3 : 1.560 s
Run 4 : 1.567 s
Run 5 : 1.540 s
Run 6 : 1.571 s
Run 7 : 1.605 s
Run 8 : 1.581 s
Run 9 : 1.550 s
Run 10: 1.586 s

Avg.  : 1.571 s
```

cmake_build_output\Perfomance_output\laptop_makefile_parallel_time_output.txt

```
Makefile (Parallel 8) build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains CMake configuration, generation and Make build time.

#######################################
# Measurement command in powershell: #
#######################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of CMake configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & cmake -B buildresults -G "Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE="cmake\toolchains\cross\STM32F103VBIx.
>> & cmake --build buildresults --parallel 8 }
# Output measured time in ms to a text file
```

```
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 2.225 s
Run 2 : 2.284 s
Run 3 : 2.204 s
Run 4 : 3.423 s
Run 5 : 3.109 s
Run 6 : 3.267 s
Run 7 : 3.257 s
Run 8 : 3.102 s
Run 9 : 3.056 s
Run 10: 3.428 s

Avg.  : 2.936 s
```

cmake_build_output\Perfomance_output\laptop_makefile_time_output.txt

```
Makefile build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains CMake configuration, generation and Make build time.

#######################################
# Measurement command in powershell: #
#######################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of cmake configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & cmake -B buildresults -G "Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE="cmake\toolchains\cross\STM32F103VBIx.
>> & cmake --build buildresults }
# Output measured time in ms to a text file
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 4.673 s
Run 2 : 4.377 s
Run 3 : 4.475 s
Run 4 : 4.278 s
Run 5 : 4.649 s
Run 6 : 4.103 s
Run 7 : 4.593 s
Run 8 : 4.536 s
Run 9 : 4.581 s
Run 10: 4.711 s

Avg.  : 4.498 s
```

cmake_build_output\Perfomance_output\laptop_ninja_time_output.txt

```
Ninja build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains CMake configuration, generation and Ninja build time

#######################################
# Measurement command in powershell: #
#######################################

# deleting previous build folder
```

```
del buildresults -Recurse -Force
# Measuring execution time of cmake configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & cmake -B buildresults -G Ninja -DCMAKE_TOOLCHAIN_FILE="cmake\toolchains\cross\STM32F103VBIx.cmake"
>> & cmake --build buildresults }
# Output measured time in ms to a text file
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 2.780 s
Run 2 : 3.211 s
Run 3 : 3.144 s
Run 4 : 3.010 s
Run 5 : 3.135 s
Run 6 : 2.600 s
Run 7 : 2.923 s
Run 8 : 3.346 s
Run 9 : 3.159 s
Run 10: 3.228 s

Avg.  : 3.054 s
```

cmake_build_output\Perfomance_output\work_laptop_makefile_parallel_time_output.txt

```
Makefile (Parallel 8) build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains CMake configuration, generation and Make build time.

######################################
# Measurement command in powershell: #
######################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of CMake configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & cmake -B buildresults -G "Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE="cmake\toolchains\cross\STM32F103VBIx.
>> & cmake --build buildresults --parallel 8 }
# Output measured time in ms to a text file
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 6.903 s
Run 2 : 6.797 s
Run 3 : 6.909 s
Run 4 : 8.268 s
Run 5 : 8.177 s
Run 6 : 8.288 s
Run 7 : 8.332 s
Run 8 : 8.206 s
Run 9 : 8.285 s
Run 10: 8.304 s

Avg. 7.847 s
```

cmake_build_output\Perfomance_output\work_laptop_makefile_time_output.txt

```
Makefile build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains CMake configuration, generation and Make build time.
```

```
####################################
# Measurement command in powershell: #
####################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of cmake configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & cmake -B buildresults -G "Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE="cmake\toolchains\cross\STM32F103VBIx.
>> & cmake --build buildresults }
# Output measured time in ms to a text file
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 9.900 s
Run 2 : 10.165 s
Run 3 : 10.058 s
Run 4 : 10.838 s
Run 5 : 10.986 s
Run 6 : 11.088 s
Run 7 : 10.977 s
Run 8 : 10.792 s
Run 9 : 11.083 s
Run 10: 9.762 s

Avg.  : 10.565 s
```

cmake_build_output\Perfomance_output\work_laptop_ninja_time_output.txt

```
Ninja build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains CMake configuration, generation and Ninja build time

####################################
# Measurement command in powershell: #
####################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of cmake configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & cmake -B buildresults -G Ninja -DCMAKE_TOOLCHAIN_FILE="cmake\toolchains\cross\STM32F103VBIx.cmake"
>> & cmake --build buildresults }
# Output measured time in ms to a text file
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 4.594 s
Run 2 : 4.553 s
Run 3 : 4.643 s
Run 4 : 4.642 s
Run 5 : 4.964 s
Run 6 : 5.556 s
Run 7 : 5.575 s
Run 8 : 5.471 s
Run 9 : 5.553 s
Run 10: 5.295 s

Avg.  : 5.085 s
```

B.4 Meson Files

meson\build\cross\STM32F103VBIx.txt

```
# Meson Cross-compilation File for STM32F103VBIx Cortex-M3 processors
# Note that Cortex-M3 does not provide an FPU
# This file should be layered after arm.txt
# Requires that arm-none-eabi-* is found in your PATH
# For more information: http://mesonbuild.com/Cross-compilation.html

[built-in options]
c_args = [ '-mcpu=cortex-m3', '-mfloat-abi=soft', '-mabi=aapcs', '-mthumb',]
c_link_args = [ '-mcpu=cortex-m3', '-mfloat-abi=soft', '-mabi=aapcs', '-mthumb',]
cpp_args = [ '-mcpu=cortex-m3', '-mfloat-abi=soft', '-mabi=aapcs', '-mthumb',]
cpp_link_args = [ '-mcpu=cortex-m3', '-mfloat-abi=soft', '-mabi=aapcs', '-mthumb']

[properties]
linker_paths = ['build/linker-scripts/stm/']
linker_scripts = ['STM32F103VBIx_FLASH.ld']
link_depends = ['build/linker-scripts/stm/STM32F103VBIx_FLASH.ld']

[host_machine]
cpu = 'STM32F103VBIx'

```

meson\build\cross\arm.txt

```
# Meson Cross-compilation File Base using GCC ARM
# Requires that arm-none-eabi-* is found in your PATH
# For more information: http://mesonbuild.com/Cross-compilation.html

[binaries]
c = 'arm-none-eabi-gcc'
cpp = 'arm-none-eabi-c++'
# *-gcc-ar is used over *-ar to support LTO flags.
# Without -gcc-ar, LTO will generate a linker warning:
# arm-none-eabi-ar: file.o: plugin needed to handle lto object
ar = 'arm-none-eabi-gcc-ar'
strip = 'arm-none-eabi-strip'

[properties]
objcopy = 'arm-none-eabi-objcopy'
# Keep this set, or the sanity check won't pass
needs_exe_wrapper = true

[host_machine]
system = 'none'
cpu_family = 'arm'
```

```
# CPU should be redefined in child cross files - this is a placeholder
# that will be used in case a child file does not override this setting
cpu = 'arm-generic'
endian = 'little'

```

meson\build\cross\cortex-m3.txt

```
# Meson Cross-compilation File for Cortex-M3 processors
# Note that Cortex-M3 does not provide an FPU
# This file should be layered after arm.txt
# Requires that arm-none-eabi-* is found in your PATH
# For more information: http://mesonbuild.com/Cross-compilation.html

[built-in options]
c_args = [ '-mcpu=cortex-m3', '-mfloat-abi=soft', '-mabi=aapcs', '-mthumb',]
c_link_args = [ '-mcpu=cortex-m3', '-mfloat-abi=soft', '-mabi=aapcs', '-mthumb',]
cpp_args = [ '-mcpu=cortex-m3', '-mfloat-abi=soft', '-mabi=aapcs', '-mthumb',]
cpp_link_args = [ '-mcpu=cortex-m3', '-mfloat-abi=soft', '-mabi=aapcs', '-mthumb',]

[host_machine]
cpu = 'cortex-m3'

```

meson\meson.build

```
project('meson_quality_test',
    ['c'],
    default_options : [
        'c_std=c11',
        'buildtype=debug',
    ],
    meson_version: '>=1.0.0',
    version: '1.0',
)

# Execute meson.build in src/
subdir('src')

```

meson\src\meson.build

```
# List of files for c_lib
c_files = files(
    'a.c',
    'b.c',
  'c.c',
    'd.c',
  'e.c',
)

# List of files for new_c_lib
```

```
11 new_c_files = files(
12   'f.c',
13   'g.c',
14   'h.c',
15   'i.c',
16   'j.c',
17 )
18
19 # Include directory for the libraries
20 c_include = include_directories('../lib')
21
22 # Create link flag variable.
23 # -nostartfiles; don't include standard startup files
24 c_link_flags = ['-nostartfiles']
25
26 # Create static library c_lib
27 c_lib = static_library(
28     'c',
29     c_files,
30     include_directories: c_include,
31 )
32
33 # Create static library new_c_lib
34 new_c_lib = static_library(
35   'new_c',
36   new_c_files,
37   include_directories: c_include,
38 )
39
40 # Dependency specifies how the files and inc should be used
41 c_dep = declare_dependency(
42     include_directories: [
43         c_include,
44     ],
45   link_args: c_link_flags,
46     link_with: [c_lib, new_c_lib],
47 )
48
49 if meson.is_cross_build()
50
51   linker_script_flags = []
52   linker_script_deps = []
53
54 # Generate linker arguments and dependencies for linker files
55
56   foreach entry : meson.get_external_property('linker_paths', [''])
57     if entry != ''
58       linker_script_flags += '-L' + meson.project_source_root() / entry
59     endif
60   endforeach
61
62   foreach entry : meson.get_external_property('linker_scripts', [''])
63     if entry != ''
64       linker_script_flags += '-T' + entry
65     endif
66   endforeach
```

```
67
68    foreach entry : meson.get_external_property('link_depends', [''])
69      if entry != ''
70        linker_script_deps += meson.project_source_root() / entry
71      endif
72    endforeach
73
74    # Map file template hardcoded for gcc compiler
75    map_file = '-Wl,-Map,@0@.map'
76
77    # Find objcopy.exe specified in the cross-file.
78    # "disabler": If "objcopy" is not found, it will short-circuit any statement using "
  host_objcopy"
79    host_objcopy = find_program(meson.get_external_property('objcopy'),
80      required: false, disabler: true)
81
82    if host_objcopy.found() == false
83      message('Specified objcopy not found, .hex and .bin conversion is disabled.')
84    endif
85
86    # Adding executable and creating .map file
87    sample_app = executable('sample_app',
88      'main.c',
89      dependencies: [c_dep],
90      link_args: [
91        linker_script_flags,
92        map_file.format(meson.current_build_dir()+'/sample_app'),
93        '-Wl,--no-gc-sections',
94      ],
95      link_depends: linker_script_deps,
96    )
97
98    # Convert output ELF to hex
99    sample_app_hex = custom_target('sample_app.hex',
100     input: sample_app,
101     output: 'sample_app.hex',
102     command: [host_objcopy, '-O', 'ihex', '@INPUT@', '@OUTPUT@'],
103     build_by_default: true
104   )
105
106   # Convert output ELF to bin
107   sample_app_bin = custom_target('sample_app.bin',
108     input: sample_app,
109     output: 'sample_app.bin',
110     command: [host_objcopy, '-O', 'binary', '@INPUT@', '@OUTPUT@'],
111     build_by_default: true
112   )
113
114 endif
```

B.5 Meson Performance Files

meson_build_output\Perfomance_output\desktop_ninja_time_output.txt

```
Ninja build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains Meson configuration, generation and Ninja build time.

#####################################
# Measurement command in powershell: #
#####################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of Meson configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & meson setup buildresults --cross-file build/cross/arm.txt --cross-file build/cross/STM32F103VBI
>> & meson compile -C buildresults }
# Output measured time in ms to a text file
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 2.703 s
Run 2 : 2.693 s
Run 3 : 2.757 s
Run 4 : 2.682 s
Run 5 : 2.690 s
Run 6 : 2.741 s
Run 7 : 2.969 s
Run 8 : 2.737 s
Run 9 : 2.756 s
Run 10: 2.708 s

Avg.  : 2.744 s
```

meson_build_output\Perfomance_output\laptop_ninja_time_output.txt

```
Ninja build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains Meson configuration, generation and Ninja build time.

#####################################
# Measurement command in powershell: #
#####################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of Meson configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & meson setup buildresults --cross-file build/cross/arm.txt --cross-file build/cross/STM32F103VBI
>> & meson compile -C buildresults }
# Output measured time in ms to a text file
```

```
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 3.007 s
Run 2 : 2.951 s
Run 3 : 2.906 s
Run 4 : 2.963 s
Run 5 : 2.824 s
Run 6 : 3.095 s
Run 7 : 2.943 s
Run 8 : 2.916 s
Run 9 : 2.973 s
Run 10: 2.818 s

avg   : 2.940 s
```

meson_build_output\Perfomance_output\work_laptop_ninja_time_output.txt

```
Ninja build time perfomance
Time measured when running a clean build 10 times.
Using powershell built-in command measurement tool.
The time measured contains Meson configuration, generation and Ninja build time.

#######################################
# Measurement command in powershell: #
#######################################

# deleting previous build folder
del buildresults -Recurse -Force
# Measuring execution time of Meson configuration, generation and Ninja build time.
>> $executionTime = Measure-Command {
>> & meson setup buildresults --cross-file build/cross/arm.txt --cross-file build/cross/STM32F103VBI
>> & meson compile -C buildresults }
# Output measured time in ms to a text file
>> $executionTime.TotalMilliseconds | Out-File -FilePath .\time.txt -Append

##################
# Cleaned output #
##################

Run 1 : 8.648 s
Run 2 : 7.716 s
Run 3 : 7.148 s
Run 4 : 7.882 s
Run 5 : 7.919 s
Run 6 : 7.678 s
Run 7 : 7.861 s
Run 8 : 8.123 s
Run 9 : 7.958 s
Run 10: 7.505 s

Avg.  : 7.844 s
```

Appendix C

Meeting Summaries

This appendix chapter contains meeting request and summaries.

C.1 Criteria meeting

The meeting was set to present a list of requirements and, together with DLSW, to expand on and add any relevant requirements. The purpose of the requirements was to develop a common understanding between the project and DLSW as to how the different build system generators should be analyzed. The meeting request can be found below, showing the requirements identified prior to the meeting.

Important topics discussed during the meeting includes:

- Build machine will only be Windows at DLSW now and in the foreseeable future.
- Reproducibility is a must-have.
- The output MUST be correct and the same at every run.
- The system must not be noticeably slower than the currently used system (Keil)
- Are the systems allowed for commercial use?

After the meeting, the final list of requirements was made and sent to DLSW for approval. The list can be found in the report at section 2.6

Meeting request for the criteria meeting on October 10.
(Original in Danish. translated to English for the report)

```
"Hello,

The meeting will be a presentation of criteria for a building system that
must be met for the system to be relevant for implementation at DLSW.
```

Following the presentation, we will engage in a discussion about the criteria and consider whether some should be removed and/or new ones added.

The outcome of this meeting will form the basis for the analysis that I will conduct on the two selected building systems (CMake and Meson).

The following is a list of my proposals for the technical capabilities that the system should have, as well as the desired qualities.

Further explanation of the qualities will be provided during the meeting.

Minor changes to the list may occur.

Capabilities:

- Compile source code to binary code
- Out-of-source builds (Build is placed in a separate build folder)
- Dependency management (installation/fetching, version checking etc.)
- Allow specification of multiple output targets within a single build system.
- Support multiple build machines and multiple compilers
- Support incremental building
- Support cross-compilation
- Version control integration

Qualities:

- Repeatability
- Reproducibility
- Standard
- Correctness
- One-touch
- Ease of use
- Performance
- Scalability
- Easy to debug
- Futureproof
- IDE compatibility "

C.2 Analysis meeting

The analysis was sent to DLSW for read-through on 31 October. A subsequent meeting was arranged and held on November 21 to discuss the content of the document. The purpose of the meeting was to give a brief overview of the analysis and use it to make a choice between the two build system generators. As the analysis had been sent before the meeting, Evaluation Matrix was presented and the main differences between the systems were discussed. this includes:

- CMake being a more mature system than Meson.
- CMake has a larger user base and is widely used in the industry.
  - Meson is a newer and smaller system with a smaller user base.
- Meson builds native and cross-build when invoked.
  - This allows unit tests and compilation for the target in the same build.
- Meson syntax is easier to use and understand.

Performance tests had not been performed prior to sending the analysis results, and these were covered during the meeting.

The meeting concluded with CMake being chosen as the build system generator to use to build the demonstrator software.

C.3 Demonstrator presentation

This meeting presented the demonstrator software. Details of this meeting and feedback from DLSW are written in the report section 5.3.

Appendix D

Demonstrator Code

This appendix chapter contains the CMake and Makefile code, performance output and directory tree for the demonstrator software.

Appendix D Contents

D.1 Main CMakeLists.txt

cmake_df3_to_print\CMakeLists.txt

```
cmake_minimum_required(VERSION 3.27 FATAL_ERROR)
cmake_policy(SET CMP0123 NEW)

project(SW03002025
    VERSION 1.00
    DESCRIPTION "DF3 project in CMake"
    LANGUAGES C ASM
)

set(DEBUG_MESSAGE_INDENT "!-----!-----!-----!-----! ")

if(CMAKE_DEBUG)
  message(DEBUG "${DEBUG_MESSAGE_INDENT}CMAKE_DEBUG ENABLED!")
endif()
message(DEBUG "${DEBUG_MESSAGE_INDENT}MESSAGE LOG LEVEL: DEBUG")

set_property(GLOBAL PROPERTY C_STANDARD 99)

###################################
# Download and Include CPM.cmake #
###################################

include(cmake/CPM.cmake)

#################################
# Include Functions and Macros #
#################################

include(buildOptions.cmake)
message(DEBUG "${DEBUG_MESSAGE_INDENT}BUILD TYPE IS: ${CMAKE_BUILD_TYPE}")
include(buildFlags.cmake)
include(cmake/fetchFunctions.cmake)
include(cmake/conversions.cmake)
include(cmake/addExecutableWithScatterFile.cmake)
include(cmake/checkAndApplyFlags.cmake)

########################
# Fetch DF3 App files #
########################

fetch_df3_app_files(V${PROJECT_VERSION_MAJOR}-${PROJECT_VERSION_MINOR})

############################
# Fetch DF3 platform files #
############################
```

```
46
47 fetch_df3_platform_files(plat_files_1 trunk 12345)
48 fetch_df3_platform_files(plat_files_2 trunk 12345)
49 # ... Truncated ...
50
51 ##################
52 # Fetch sysCore #
53 ##################
54
55 fetch_clear_syscore(sys_1 trunk 12345)
56 fetch_clear_syscore(sys_2 trunk 12345)
57 # ... Truncated ...
58
59 ########################
60 # Fetch External files #
61 ########################
62
63 fetch_extlib(extlib_1 trunk 12345)
64 fetch_extlib(extlib_2 trunk 12345)
65 # ... Truncated ...
66
67 ########################
68 # Fetch CLEAR Drivers #
69 ########################
70
71 fetch_clear_driver(driver_1 trunk 12345)
72 fetch_clear_driver(driver_2 trunk 12345)
73 # ... Truncated ...
74
75 ##############################
76 # Fetch CLEAR Common Modules #
77 ##############################
78
79 fetch_clear_common(common_1 trunk 12345)
80 fetch_clear_common(common_2 trunk 12345)
81 # ... Truncated ...
82
83 ########################
84 # Fetch CLEAR Modules #
85 ########################
86
87 fetch_clear_module(module_1 trunk 12345)
88 fetch_clear_module(module_2 trunk 12345)
89 # ... Truncated ...
90 fetch_clear_module(module_n trunk 12345)
91
92 ########################
93 # Process Source Tree #
94 ########################
95
96 add_subdirectory(${CMAKE_SOURCE_DIR}/src)
```

D.2 Source CMakeLists.txt

cmake_df3_to_print\src\CMakeLists.txt

```
set(exe_name ${PROJECT_NAME})
add_executable(${exe_name})

################################################################
# Add compile and link flags defined in ../buildFlags.cmake #
################################################################

add_exe_compile_flags(${exe_name} PRIVATE)
add_exe_build_type_compile_flags(${exe_name} PRIVATE)
add_exe_linker_flags(${exe_name} PRIVATE)

########################################
# Output, CRC and memory map function #
########################################

convert_to_hex(${exe_name})
crc32(${exe_name})
target_linker_map(${exe_name})

################
# SOURCE FILES #
################

# startup STM32g030xx, DF3
set_property(SOURCE ${df3_app_SOURCE_DIR}/startup_boot_stm32g030xx.s PROPERTY LANGUAGE ASM)
set_property(SOURCE ${df3_app_SOURCE_DIR}/startup_stm32g030xx.s PROPERTY LANGUAGE ASM)
target_sources(${exe_name}
  PRIVATE
    ${df3_app_SOURCE_DIR}/startup_boot_stm32g030xx.s
    #${df3_app_SOURCE_DIR}/startup_stm32g030xx.s
)

# module_1
target_sources(${exe_name} PRIVATE
# ... Redacted ...
)

# module_2
target_sources(${exe_name} PRIVATE
# ... Redacted ...
)

# module_3
target_sources(${exe_name} PRIVATE
# ... Redacted ...
```

```
46 )
47
48 #        ...
49 # ... Truncated ...
50 #        ...
51
52 # module_n
53 target_sources(${exe_name} PRIVATE
54 # ... Redacted ...
55 )
56
57 target_include_directories(${exe_name}
58   PRIVATE
59   # ... Redacted ...
60 )
61
62 if(DEFINED SOURCE_FILE AND DEFINED COMPILE_FLAG)
63   message(DEBUG "${DEBUG_MESSAGE_INDENT}SOURCE_FILE: ${SOURCE_FILE} --- COMPILE_FLAG:
   ${COMPILE_FLAG}")
64   file(GLOB_RECURSE TARGET_FILE "${FETCHCONTENT_BASE_DIR}/${SOURCE_FILE}")
65   message(DEBUG "${DEBUG_MESSAGE_INDENT}MY_FILE_PATH/SOURCE_FILE: ${TARGET_FILE}")
66   set_source_files_properties(${TARGET_FILE} PROPERTIES COMPILE_FLAGS "${COMPILE_FLAG}")
67 endif()
```

D.3 Makefile Interface

cmake_df3_to_print\Makefile

```
# you can set this to 1 to see all commands that are being run
VERBOSE ?= 0

ifeq ($(VERBOSE),1)
export Q :=
export VERBOSE := 1
else
export Q := @
export VERBOSE := 0
endif

# This skeleton is built for CMake's Ninja generator
export CMAKE_GENERATOR=Ninja

BUILDRESULTS ?= buildresults
CONFIGURED_BUILD_DEP = $(BUILDRESULTS)/build.ninja

OPTIONS ?=
INTERNAL_OPTIONS =

LTO ?= 0
ifneq ($(LTO),0)
  INTERNAL_OPTIONS += -DENABLE_LTO=ON
endif

CROSS ?= STM32G03x
ifneq ($(CROSS),)
  INTERNAL_OPTIONS += -DCMAKE_TOOLCHAIN_FILE=cmake/toolchains/$(CROSS).cmake
endif

BUILD_TYPE ?=
ifneq ($(BUILD_TYPE),)
  INTERNAL_OPTIONS += -DCMAKE_BUILD_TYPE=$(BUILD_TYPE)
endif

LOG_LEVEL ?=
ifneq ($(LOG_LEVEL),)
  INTERNAL_OPTIONS += -DCMAKE_MESSAGE_LOG_LEVEL=${LOG_LEVEL}
endif

WNO_DEV ?= 0
ifneq ($(WNO_DEV),0)
  INTERNAL_OPTIONS += -Wno-dev
endif

CMAKE_DEBUG ?= 0
ifneq ($(CMAKE_DEBUG),0)
  INTERNAL_OPTIONS += -DCMAKE_DEBUG=1
  INTERNAL_OPTIONS += -DCMAKE_MESSAGE_LOG_LEVEL=DEBUG
endif

CPM_DEBUG ?= 0
ifneq ($(CPM_DEBUG),0)
```

```
54    INTERNAL_OPTIONS += -DCPM_DEBUG=1
55  endif
56
57  SOURCE_FILE ?=
58  ifneq ($(SOURCE_FILE),)
59    INTERNAL_OPTIONS += -DSOURCE_FILE=${SOURCE_FILE}
60  endif
61
62  COMPILE_FLAG ?=
63  ifneq ($(COMPILE_FLAG),)
64    INTERNAL_OPTIONS += -DCOMPILE_FLAG=${COMPILE_FLAG}
65  endif
66
67  all: default
68
69  .PHONY: default
70  default: | $(CONFIGURED_BUILD_DEP)
71    $(Q)ninja -C $(BUILDRESULTS)
72
73  # Runs whenever the build has not been configured successfully
74  $(CONFIGURED_BUILD_DEP):
75    $(Q)cmake -B $(BUILDRESULTS) $(OPTIONS) $(INTERNAL_OPTIONS)
76
77  # Manually Reconfigure a target, esp. with new options
78  .PHONY: reconfig
79  reconfig:
80    $(Q) cmake -B $(BUILDRESULTS) $(OPTIONS) $(INTERNAL_OPTIONS) --fresh
81
82  .PHONY: clean
83  clean:
84    $(Q) powershell -Command "if (Test-Path -Path "C:\thesis\cmake_df3\$(BUILDRESULTS)") {ninja
    -C $(BUILDRESULTS) clean} \
85    else{echo '-- Cannot clean build artifacts: Build result directory not found'}"
86
87  .PHONY: distclean
88  # Deletes the configured build output directory
89  distclean:
90    $(Q) powershell -Command "Remove-Item -Recurse -Force $(BUILDRESULTS)"
91
92  .PHONY : help
93  help :
94    @echo "
95    @echo " Usage: make [OPTIONS] <target>
96    @echo "
97    @echo "  Options:
98    @echo "    > VERBOSE Show verbose output for Make rules. Default 0. Enable with 1.
99    @echo "    > BUILDRESULTS Directory for build results. Default buildresults.
100   @echo "    > OPTIONS Configuration options to pass to a build. Default empty.
101   @echo "    > LTO Enable LTO builds. Default 0. Enable with 1.
102   @echo "      LTO is enabled by default for "Release" builds.
103   @echo "    > CROSS Enable a Cross-compilation build. (Default "STM32G03x")
104   @echo "      Pass the cross-compilation toolchain name without a path or extension.
105   @echo "      Example: make CROSS=cortex-m0plus
106   @echo "      For supported toolchains, see cmake/toolchains/
107   @echo "    > BUILD_TYPE Specify the build type (default: Release)
108   @echo "      Options are: Debug Release MinSizeRel RelWithDebInfo
```

```
109   @echo "    > LOG_LEVEL Specify the logging level of messages (default: STATUS)
110   @echo "      Options are: STATUS DEBUG
111   @echo "    > WNO_DEV supress -Wno-dev warnings. Default 0. Enable with 1.
112   @echo "    > CMAKE_DEBUG enables CMake debug functions and messages
113   @echo "      Default 0. Enable with 1.
114   @echo "    > CPM_DEBUG enables CPM debug messages. Default 0. Enable with 1.
115   @echo "    > SOURCE_FILE Source file target for the COMPILE_FLAG option.
116   @echo "    > COMPILE_FLAG Specify compile flag(s) for SOURCE_FILE.
117   @echo "      Example: make SOURCE_FILE="my_file.c" COMPILE_FLAG="-O1"
118   @echo "
119   @echo "  Targets:
120   @echo "    default: Builds all default targets ninja knows about
121   @echo "    clean: cleans build artifacts, keeping build files in place
122   @echo "    distclean: deletes the configured build output directory
123   @echo "    reconfig: Reconfigure an existing build output folder with new settings
124   @echo "
```

D.4 Toolchain Files

cmake\toolchains\STM32G03x.cmake

```
############################
# STM32G03x (Cortex-M0+) #
############################

# CMake includes the toolchain file multiple times when configuring the build,
# which causes errors for some flags. This is prevented with an include guard.
if(STM32G03x_TOOLCHAIN_INCLUDED)
  return()
endif()

set(STM32G03x_TOOLCHAIN_INCLUDED TRUE)

set(CPU_NAME STM32G03x)

set(SCATTER_FILE_DIR ${CMAKE_SOURCE_DIR}/cmake/scatterFiles)

set(CPU_FLAGS "-DSTM32G030xx -D__MICROLIB")
set(ASM_FLAGS "--pd \"__MICROLIB SETA 1\"")
set(LD_FLAGS "--library_type=microlib")

include(${CMAKE_CURRENT_LIST_DIR}/cortex-m0plus.cmake)
```

cmake\toolchains\armcc.cmake

```
####################################
# armcc Base Toolchain #
####################################
#
# To include this file as a base toolchain file,
# include it at the bottom of the derived toolchain file.
#
# You can define CPU_FLAGS that will be passed to CMAKE_*_FLAGS to select the CPU
# (and any other necessary CPU-specific flags)
# You can define LD_FLAGS to control linker flags for your target

#################
# System Config #
#################

set(CMAKE_SYSTEM_NAME Generic)
if(NOT CMAKE_SYSTEM_PROCESSOR)
  set(CMAKE_SYSTEM_PROCESSOR arm)
endif()
# Represents the name of the specific processor type, e.g. Cortex-M0plus
```

```
21 if(NOT CPU_NAME)
22   set(CPU_NAME generic)
23 endif()
24
25 ####################
26 # Toolchain Config #
27 ####################
28
29 set(CMAKE_C_COMPILER    C:/Keil_v5/ARM/ARM_Compiler_5.06u5/bin/armcc.exe)
30 set(CMAKE_ASM_COMPILER  C:/Keil_v5/ARM/ARM_Compiler_5.06u5/bin/armasm.exe)
31 set(CMAKE_AR            C:/Keil_v5/ARM/ARM_Compiler_5.06u5/bin/armar.exe)
32 set(OBJCOPY             C:/Keil_v5/ARM/ARM_Compiler_5.06u5/bin/fromelf.exe)
33
34 # Test compiles will use static libraries
35 set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)
36
37 ################
38 # Common Flags #
39 ################
40 # Note that CPU_FLAGS, ASM_FLAGS, and LD_FLAGS are set by other Toolchain files
41 # that include this file.
42 #
43 # See the CMake Manual for CMAKE_<LANG>_FLAGS_INIT:
44 # https://cmake.org/cmake/help/latest/variable/CMAKE_LANG_FLAGS_INIT.html
45
46 include(${CMAKE_CURRENT_LIST_DIR}/overrideBuildTypeSettings.cmake)
47
48 set(CMAKE_C_FLAGS_INIT
49   "${CPU_FLAGS}"
50   CACHE
51   INTERNAL "Default C compiler flags.")
52 set(CMAKE_ASM_FLAGS_INIT
53   "${ASM_FLAGS}"
54   CACHE
55   INTERNAL "Default ASM compiler flags.")
56 set(CMAKE_EXE_LINKER_FLAGS_INIT
57   "${LD_FLAGS}"
58   CACHE
59   INTERNAL "Default linker flags.")
```

cmake\toolchains\armclang.cmake

```
1  ###################################
2  # armclang Base Toolchain #
3  ###################################
4  #
5  # To include this file as a base toolchain file,
6  # include it at the bottom of the derived toolchain file.
7  #
8  # You can define CPU_FLAGS that will be passed to CMAKE_*_FLAGS to select the CPU
9  # (and any other necessary CPU-specific flags)
10 # You can define LD_FLAGS to control linker flags for your target
11
12 ##################
13 # System Config #
```

```
14 ##################
15
16 set(CMAKE_SYSTEM_NAME Generic)
17 if(NOT CMAKE_SYSTEM_PROCESSOR)
18   set(CMAKE_SYSTEM_PROCESSOR arm)
19 endif()
20 # Represents the name of the specific processor type, e.g. Cortex-M0plus
21 if(NOT CPU_NAME)
22   set(CPU_NAME generic)
23 endif()
24
25 ####################
26 # Toolchain Config #
27 ####################
28
29 set(CMAKE_C_COMPILER    armclang)
30 set(CMAKE_ASM_COMPILER  armasm)
31 set(CMAKE_AR            armar)
32 set(OBJCOPY             fromelf)
33
34 # Test compiles will use static libraries
35 set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)
36
37 ################
38 # Common Flags #
39 ################
40 # Note that CPU_FLAGS, ASM_FLAGS, and LD_FLAGS are set by other Toolchain files
41 # that include this file.
42 #
43 # See the CMake Manual for CMAKE_<LANG>_FLAGS_INIT:
44 # https://cmake.org/cmake/help/latest/variable/CMAKE_LANG_FLAGS_INIT.html
45
46 include(${CMAKE_CURRENT_LIST_DIR}/overrideBuildTypeSettings.cmake)
47 set(CMAKE_C_FLAGS_INIT
48   "${CPU_FLAGS}"
49   CACHE
50   INTERNAL "Default C compiler flags.")
51 set(CMAKE_ASM_FLAGS_INIT
52   "${ASM_FLAGS}"
53   CACHE
54   INTERNAL "Default ASM compiler flags.")
55 set(CMAKE_EXE_LINKER_FLAGS_INIT
56   "${LD_FLAGS}"
57   CACHE
58   INTERNAL "Default linker flags.")
```

cmake\toolchains\cortex-m0plus.cmake

```
1 ##############
2 # Cortex-M0+ #
3 ##############
4
5 # CMake includes the toolchain file multiple times when configuring the build,
6 # which causes errors for some flags. This is prevented with an include guard.
7 if(ARM_CORTEX_M0+_TOOLCHAIN_INCLUDED)
```

```
  return()
endif()

set(ARM_CORTEX_M0+_TOOLCHAIN_INCLUDED TRUE)

if(NOT CPU_NAME)
  set(CPU_NAME cortex-m0plus)
endif()

set(CMAKE_SYSTEM_PROCESSOR cortex-m0plus)
set(CMAKE_SYSTEM_ARCH armv6-m)

set(CPU_FLAGS
  "-mcpu=cortex-m0plus \
  -march=armv6-m \
  ${CPU_FLAGS}")
set(ASM_FLAGS
  "--cpu Cortex-M0+ \
  ${ASM_FLAGS}")
set(LD_FLAGS
  "${LD_FLAGS} \
  --cpu=Cortex-M0+" )

include(${CMAKE_CURRENT_LIST_DIR}/armclang.cmake)
# include(${CMAKE_CURRENT_LIST_DIR}/armcc.cmake)
```

cmake\toolchains\overrideBuildTypeSettings.cmake

```
#####################################
# Default Build Configration Flags #
#####################################

set(CMAKE_C_FLAGS_DEBUG
    "-Oz -DDEBUG -DDEBUG_FAULT_DIAGNOSTICS -DEXPOSE_3DIGIT_ERR_CODES"
    CACHE
    STRING "Default C compiler debug build flags."
)
set(CMAKE_ASM_FLAGS_DEBUG
    "-g --xref"
    CACHE
    STRING "Default ASM assembler debug build flags."
)

# Release flags
set(CMAKE_C_FLAGS_RELEASE
    "-Oz"
    CACHE
    STRING "Default C compiler release build flags."
)
set(CMAKE_ASM_FLAGS_RELEASE
    ""
    CACHE
    STRING "Default ASM assembler release build flags."
)
```

D.5 Functions, Options and Flags

cmake\CPM.cmake

```
set(CPM_DOWNLOAD_VERSION 0.38.3)
set(CPM_DOWNLOAD_LOCATION "${CMAKE_BINARY_DIR}/cmake/CPM_${CPM_DOWNLOAD_VERSION}.cmake")

# Expand relative path. this is important if the provided path contains a tilde(~)
get_filename_component(CPM_DOWNLOAD_LOCATION ${CPM_DOWNLOAD_LOCATION} ABSOLUTE)

function(download_cpm)
    message(STATUS "Downloading CPM")
    file(
    DOWNLOAD
        https://github.com/cpm-cmake/CPM.cmake/releases/download/v${CPM_DOWNLOAD_VERSION}
/CPM.cmake
        ${CPM_DOWNLOAD_LOCATION}
        EXPECTED_HASH SHA256=cc155ce02e7945e7b8967ddfaff0b050e958a723ef7aad3766d368940cb15494
    )
endfunction()

download_cpm()
include(${CPM_DOWNLOAD_LOCATION})
```

cmake\addExecutableWithScatterFile.cmake

```
function(add_executable TARGET)
  _add_executable(${TARGET} ${ARGN})
  if(SCATTER_FILE_DIR)
    string(TOLOWER ${CMAKE_BUILD_TYPE} build_type)
    set_target_properties(${TARGET} PROPERTIES
      LINK_DEPENDS "${SCATTER_FILE_DIR}/${build_type}.sct"
    )
    target_link_options(${TARGET} PRIVATE
        "--scatter"
        "${SCATTER_FILE_DIR}/${build_type}.sct"
    )
    endif()
endfunction()
```

cmake\checkAndApplyFlags.cmake

```
########################################
# Functions to check if compiler flag #
#   is supported by the used compiler #
########################################

```

```
6  macro(make_safe_varname input_string output_var)
7    STRING(REPLACE "-" "_" ${output_var} ${input_string})
8  endmacro()
9
10 ######################
11 # C Language Support #
12 ######################
13
14 function(apply_supported_c_compiler_flag target scope flag)
15   if (NOT DEFINED CheckCCompilerFlag_INCLUDED)
16     include(CheckCCompilerFlag)
17     set(CheckCCompilerFlag_INCLUDED TRUE)
18   endif()
19
20   make_safe_varname(${flag} flag_var)
21   check_c_compiler_flag(${flag} ${flag_var})
22
23   if(${flag_var})
24     target_compile_options(${target} ${scope} $<$<COMPILE_LANGUAGE:C>:${flag}>)
25   endif()
26 endfunction()
27
28 ############################
29 # Add exe build type flags #
30 ############################
31
32 function(add_exe_build_type_compile_flags target scope)
33   string(TOLOWER ${CMAKE_BUILD_TYPE} build_type_lc)
34   foreach(flag ${exe_${build_type_lc}_compile_flags})
35     target_compile_options(${target} ${scope} $<$<COMPILE_LANGUAGE:C>:${flag}>)
36   endforeach()
37 endfunction()
38
39 ########################
40 # Add exe Linker flags #
41 ########################
42
43 function(add_exe_linker_flags target scope)
44   foreach(flag ${exe_linker_flags})
45     target_link_options(${target} ${scope} ${flag})
46   endforeach()
47 endfunction()
48
49 ##########################
50 # Add exe compiler flags #
51 ##########################
52
53 function(add_exe_compile_flags target scope)
54   foreach(flag ${exe_compile_flags})
55     if(${CMAKE_DEBUG})
56       apply_supported_c_compiler_flag(${target} ${scope} ${flag})
57     else()
58       target_compile_options(${target} ${scope} $<$<COMPILE_LANGUAGE:C>:${flag}>)
59     endif()
60   endforeach()
61 endfunction()
```

cmake\conversions.cmake

```
function(convert_to_hex TARGET)
    add_custom_command(
        TARGET ${TARGET} POST_BUILD
        COMMAND ${OBJCOPY} --i32combined --output=${TARGET}V${PROJECT_VERSION_MAJOR}-${PROJECT_VERSION_MINOR}.hex ${TARGET}.elf
        BYPRODUCTS ${TARGET}V${PROJECT_VERSION_MAJOR}-${PROJECT_VERSION_MINOR}.hex
    )
endfunction()

function(crc32 TARGET)
  add_custom_command(
      TARGET ${TARGET} POST_BUILD
      COMMAND ${CMAKE_SOURCE_DIR}/crc32/crc32.exe ${TARGET}V${PROJECT_VERSION_MAJOR}-${PROJECT_VERSION_MINOR}.hex ${CMAKE_SOURCE_DIR}/crc32/crc32.ini
  )
endfunction()

function(target_linker_map TARGET)
    target_link_options(${TARGET} PRIVATE
        "--map"
        "--load_addr_map_info"
        "--xref"
        "--callgraph"
        "--symbols"
        "SHELL:--info sizes"
        "SHELL:--info totals"
        "SHELL:--info unused"
        "SHELL:--info veneers"
    )
endfunction()
```

cmake\fetchFunctions.cmake

```
if(NOT ${CPM_DEBUG})
  function(cpm_message)
  endfunction()
endif()

function(add_spaces num)
    set(spaces "")
    foreach(_ RANGE 1 ${num})
        set(spaces "${spaces} ")
    endforeach()

    set(${CMAKE_CURRENT_FUNCTION} "${spaces}" PARENT_SCOPE)
endfunction()

function(append_spaces input)
    set(space_appended_input ${input})
    string(LENGTH ${space_appended_input} string_length)
    while(${string_length} LESS 20)
        set(space_appended_input "${space_appended_input} ")
```

```
20          string(LENGTH ${space_appended_input} string_length)
21      endwhile()
22      set(${CMAKE_CURRENT_FUNCTION} "${space_appended_input}" PARENT_SCOPE)
23  endfunction()
24
25  function(check_content_added input)
26    if(${input}_ADDED)
27      message(CHECK_PASS "${input} - FOUND")
28    else()
29      message(CHECK_FAIL "${input} - NOT FOUND")
30    endif()
31  endfunction()
32
33  macro(fetch_clear_driver TYPE VERSION REVISION)
34      append_spaces(${TYPE})
35      message(CHECK_START "Fetching CLEAR driver: ${append_spaces}:${VERSION} @ ${REVISION}")
36      if(${TYPE} STREQUAL stm32g03x)
37          set(NAME drv_stm32g03x)
38          set(URL ${SVN_REPO}/CLEAR/drv/st/stm32g03x/base/${VERSION}/src)
39      elseif(${TYPE} STREQUAL st_common)
40          set(NAME drv_st_common)
41          set(URL ${SVN_REPO}/CLEAR/drv/st/common/${VERSION}/lib)
42      elseif(${TYPE} STREQUAL if)
43          set(NAME drv)
44          set(URL ${SVN_REPO}/CLEAR/drv/if/base/${VERSION}/if)
45      elseif(${TYPE} STREQUAL common)
46          set(NAME drv_common)
47          set(URL ${SVN_REPO}/CLEAR/drv/common/base/${VERSION}/lib)
48      else()
49          message(FATAL_ERROR "${TYPE} DOES NOT EXIST")
50      endif()
51      CPMAddPackage(
52          NAME ${NAME}
53          SVN_REPOSITORY ${URL}
54          DOWNLOAD_ONLY ON
55          SVN_REVISION -r${REVISION}
56      )
57      if(${CMAKE_DEBUG})
58        check_content_added(${NAME})
59      endif()
60      unset(NAME)
61      unset(URL)
62  endmacro()
63
64  macro(fetch_clear_common MODULE_NAME VERSION REVISION)
65      append_spaces(${MODULE_NAME})
66      message(CHECK_START "Fetching CLEAR common: ${append_spaces}:${VERSION} @ ${REVISION}")
67      CPMAddPackage(
68          NAME ${MODULE_NAME}
69          SVN_REPOSITORY ${SVN_REPO}/CLEAR/common/${MODULE_NAME}/${VERSION}/lib
70          DOWNLOAD_ONLY ON
71          SVN_REVISION -r${REVISION}
72      )
73      if(${CMAKE_DEBUG})
74        check_content_added(${MODULE_NAME})
75      endif()
```

```
76  endmacro()
77
78  macro(fetch_clear_module MODULE_NAME VERSION REVISION)
79      append_spaces(${MODULE_NAME})
80      message(CHECK_START "Fetching CLEAR module: ${append_spaces}:${VERSION} @ ${REVISION}")
81      CPMAddPackage(
82          NAME ${MODULE_NAME}
83          SVN_REPOSITORY ${SVN_REPO}/CLEAR/mod/${MODULE_NAME}/${VERSION}/lib
84          DOWNLOAD_ONLY ON
85          SVN_REVISION -r${REVISION}
86      )
87      if(${CMAKE_DEBUG})
88        check_content_added(${MODULE_NAME})
89      endif()
90  endmacro()
91
92  macro(fetch_clear_syscore TYPE VERSION REVISION)
93      if(${TYPE} STREQUAL "st")
94          append_spaces("st_common")
95          add_spaces(5)
96          message(CHECK_START "Fetching sysCore${add_spaces}: ${append_spaces}:${VERSION} @
    ${REVISION}")
97          set(NAME syscore_st)
98          set(URL ${SVN_REPO}/CLEAR/sysCore/st/common/${VERSION}/src)
99      elseif(${TYPE} STREQUAL "common")
100         append_spaces("common")
101         add_spaces(5)
102         message(CHECK_START "Fetching sysCore${add_spaces}: ${append_spaces}:${VERSION} @
    ${REVISION}")
103         set(NAME syscore_common)
104         set(URL ${SVN_REPO}/CLEAR/sysCore/common/${VERSION}/lib)
105     elseif(${TYPE} STREQUAL "if")
106         append_spaces("interface")
107         add_spaces(5)
108         message(CHECK_START "Fetching sysCore${add_spaces}: ${append_spaces}:${VERSION} @
    ${REVISION}")
109         set(NAME syscore)
110         set(URL ${SVN_REPO}/CLEAR/sysCore/if/${VERSION}/if)
111     else()
112         message(FATAL_ERROR "${TYPE} DOES NOT EXIST")
113     endif()
114     CPMAddPackage(
115             NAME ${NAME}
116             SVN_REPOSITORY ${URL}
117             DOWNLOAD_ONLY ON
118             SVN_REVISION -r${REVISION}
119     )
120     if(${CMAKE_DEBUG})
121       check_content_added(${NAME})
122     endif()
123     unset(NAME)
124     unset(URL)
125 endmacro()
126
127 macro(fetch_extlib EXTLIB_NAME EXTLIB_VERSION REVISION)
128     append_spaces(${EXTLIB_NAME})
```

```
129     message(CHECK_START "Fetching external lib: ${append_spaces}:${EXTLIB_VERSION} @
    ${REVISION}")
130     CPMAddPackage(
131         NAME ${EXTLIB_NAME}
132         SVN_REPOSITORY ${SVN_REPO}/extlib/${EXTLIB_NAME}/${EXTLIB_VERSION}
133         DOWNLOAD_ONLY ON
134         SVN_REVISION -r${REVISION}
135     )
136     if(${CMAKE_DEBUG})
137       check_content_added(${EXTLIB_NAME})
138     endif()
139 endmacro()
140
141 macro(fetch_df3_platform_files TYPE VERSION REVISION)
142     if(${TYPE} STREQUAL "cfg")
143     append_spaces(${TYPE})
144     message(CHECK_START "Fetching DF3 platform: ${append_spaces}:${VERSION} @ ${REVISION}")
145     set(NAME df3_cfg)
146     set(URL ${SVN_REPO}/platforms/DF3/cfg/${VERSION})
147     elseif(${TYPE} STREQUAL "runtimeDiag")
148     append_spaces(${TYPE})
149     message(CHECK_START "Fetching DF3 platform: ${append_spaces}:${VERSION} @ ${REVISION}")
150     set(NAME df3_runtimeDiag)
151     set(URL ${SVN_REPO}/platforms/DF3/mod/runtimeDiag/${VERSION}/lib)
152     else()
153         message(FATAL_ERROR "${TYPE} DOES NOT EXIST")
154     endif()
155     CPMAddPackage(
156         NAME ${NAME}
157         SVN_REPOSITORY ${URL}
158         DOWNLOAD_ONLY ON
159         SVN_REVISION -r${REVISION}
160     )
161     if(${CMAKE_DEBUG})
162       check_content_added(${NAME})
163     endif()
164     unset(NAME)
165     unset(URL)
166 endmacro()
167
168 macro(fetch_df3_app_files VERSION)
169     add_spaces(12)
170     message(CHECK_START "Fetching DF3 application files:${add_spaces}:${VERSION}")
171     CPMAddPackage(
172         NAME df3_app
173         SVN_REPOSITORY ${SVN_REPO}/platforms/DF3/app/3002025/tags/released/${VERSION}
    /application
174         DOWNLOAD_ONLY ON
175     )
176     if(${CMAKE_DEBUG})
177       check_content_added(df3_app)
178     endif()
179 endmacro()
180
```

cmake_df3_to_print\buildOptions.cmake

```
1  ##############################################
2  # Default Settings for CMake Cache Variables #
3  ##############################################
4
5  set(default_build_type "Release")
6  if(NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
7      message(STATUS "Setting build type to '${default_build_type}' as none was specified")
8      set(CMAKE_BUILD_TYPE "${default_build_type}" CACHE
9          STRING "Choose type of build"
10         FORCE
11     )
12     # Set the possible values of build type for cmake-gui
13     set_property(CACHE CMAKE_BUILD_TYPE
14         PROPERTY STRINGS "Debug" "Release"
15     )
16 endif()
17
18 ####################
19 # Project Options #
20 ####################
21
22 include(CheckIPOSupported)
23
24 check_ipo_supported(RESULT lto_supported)
25 if("${lto_supported}")
26     option(ENABLE_LTO
27     "Enable link-time optimization"
28     OFF)
29 endif()
30
31 if("${ENABLE_LTO}" OR "${CMAKE_BUILD_TYPE}" STREQUAL "Release")
32   message(DEBUG "${DEBUG_MESSAGE_INDENT}LTO IS ENABLED")
33   string(APPEND CMAKE_C_FLAGS " -flto")
34   string(APPEND CMAKE_EXE_LINKER_FLAGS " --lto")
35 endif()
36
37 option(CMAKE_DEBUG
38   "Enable various CMake debug functions (e.g. compile flag check)"
39   OFF)
40
41 option(CPM_DEBUG
42   "Enable CPM messages"
43   OFF)
44
```

```
45 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
```

cmake_df3_to_print\buildFlags.cmake

```
1
2  list(APPEND exe_compile_flags
3    "-xc"
4    "-c"
5    "-std=gnu99"
6  # ... Truncated ...
7  )
8
9  list(APPEND exe_linker_flags
10   "--strict"
11   "--summary_stderr"
12   "--info" "summarysizes"
13 # ... Truncated ...
14 )
15
16 list(APPEND exe_release_compile_flags
17   "-DSOFTWARE_VERSION=1"
18   "-DSOFTWARE_NUMBER=1"
19 # ... Truncated ...
20 )
21
22 list(APPEND exe_debug_compile_flags
23   "-DSOFTWARE_VERSION=2"
24   "-DSOFTWARE_NUMBER=2"
25 # ... Truncated ...
26 )
```

D.6 Performance Output

Keil_build_time.txt

```
Keil build time for DF3 project
Time was measured when running a clean build 10 times.
The average was calculated from the build time output given by Keil.
The time output could only be provided with whole second accuracy.

The build was initiated through Keil by pressing the "rebuild" button.

Run 1 : 43 s
Run 2 : 42 s
Run 3 : 41 s
Run 4 : 40 s
Run 5 : 41 s
Run 6 : 41 s
Run 7 : 41 s
Run 8 : 42 s
Run 9 : 41 s
Run 10: 41 s

Avg. 41.3 s
```

CMake_ninja_build_time.txt

```
CMake (Ninja) build time for DF3 project
Time was measured when running a clean build 10 times.
The average was calculated from the output time measured through PowerShell

The build was invoked and measured with the following PowerShell commands

PS C:\thesis\cmake_df3\buildresults> ninja clean
>> $executionTime = Measure-Command { ninja }
>> $executionTime.TotalMilliseconds | Out-File -FilePath ..\time_output.txt -Append

Run 1 : 20.401 s
Run 2 : 21.703 s
Run 3 : 23.873 s
Run 4 : 23.468 s
Run 5 : 23.951 s
Run 6 : 23.853 s
Run 7 : 23.404 s
Run 8 : 23.717 s
Run 9 : 23.497 s
Run 10: 23.286 s

Avg.  : 23.116 s
```

D.7 Directory Tree

```
DF3
├── cmake
│   ├── toolchains
│   │   ├── armclang.cmake
│   │   ├── cortex-m0plus.cmake
│   │   ├── STM32G03x.cmake
│   │   └── overrideBuildTypeSettings.cmake
│   ├── addExecutableWithScatterFile.cmake
│   ├── checkAndApplyFlags.cmake
│   ├── conversions.cmake
│   ├── CPM.cmake
│   └── fetchFunctions.cmake
├── crc32
│   ├── crc32.exe
│   └── crc32.ini
├── scatterFiles
│   ├── debug.sct
│   └── release.sct
├── src
│   └── CMakeLists.txt
├── buildFlags.cmake
├── buildOptions.cmake
├── CMakeLists.txt
└── Makefile
```