

**Project “Neo CdM-16 LLVM back-end”
:: Software requirements specification**

Andrey Eliseev, Matvey Zenin, Stepan Kuchenkov, Alexander Volk

Contents

1. Introduction	3
2. Stakeholders	3
3. Glossary	3
4. Actors	3
5. Functional requirements	4
5.1. Use cases for Clang users	4
5.1.1. Input files	5
5.1.2. Output files	5
5.1.3. Supported options	5
5.2. Use cases for Rustc users	6
5.2.1. Input files	7
5.2.2. Output files	7
5.3. Common use cases	7
6. Non-functional requirements	8
6.1. Environment	8
6.2. Language support limitations	8
6.2.1. ISO/GNU C Language	8
6.2.2. Rust programming language	8
6.3. Extensibility	8

1. Introduction

Our project is about revisiting, improving, and bringing the CdM-16 LLVM code generation backend and compiler toolchain to a more usable state. The goal is to enable the compilation of C (via clang) and Rust (via rustc) source code into linkable object files for the CdM-16 target without any workarounds, out of the box. The existing Clang solution can only be used to generate assembly code, but not as a fully-fledged compilation driver.

Other than that, we are going to focus on fixing existing issues and improving CdM-16 target code generation.

2. Stakeholders

- *Maintainers*: developers handling the project.
- *Toolchain users (Students)*: the primary end users who will use the toolchain to develop projects targeting the CdM-16 platform.
- *Educators*: rely on the toolchain working properly to teach programming on the CdM-16 platform.

3. Glossary

- LLVM - open-source project providing a collection of modular and reusable compiler and toolchain technologies (tools, libraries, DSL, etc.).
- Compilation driver - the top-level program that orchestrates the compilation process. It invokes the frontend, backend, linker and assembler as needed.
- Compiler frontend - the part of the compiler that processes source code and generates intermediate representation (IR) code.
- Compiler backend - the part of the compiler that translates the IR into target-specific assembly or machine code.

4. Actors

Actor	Description
Clang user	A person or a build tool (e.g. <code>make</code>) using the <code>clang</code> program
Rustc user	A person or a build tool (e.g. <code>cargo</code>) using the <code>rustc</code> program

5. Functional requirements

5.1. Use cases for Clang users

Use case "compile with clang"

The usage of the CdM LLVM backend for compiling C code with the Clang compiler.

Actor: Clang user

Goals: To process one or many input files and produce one or many output files (see “Input files” and “Output files” sections).

Precondition: The user has one or more input files.

Main success scenario

Steps:

1. User executes the program providing the necessary options and the names of the input files in command line arguments.
2. The program prints any warning messages to `stderr`.
3. The program writes one or more output files.
4. The program exits with code 0.

Alternative scenario "compilation error"

Conditions - any of:

- The provided combination of options is not supported (see “Supported options” section).
- Some of the input files do not exist at the specified locations.
- Some of the input files are invalid (see “Input files” section).
- No input files were specified.
- The compiler was not able to invoke the `cocas` executable.

Steps:

1. User executes the program providing the necessary options and the names of the input files in command line arguments.
2. The program prints the error and warning messages to `stderr`.
3. The program writes zero or more output files not affected by the errors.
4. The program exits with some code other than 0.

5.1.1. Input files

The compiler shall accept any combination of files of the following types:

- C source files, as specified in **ISO/IEC 9899**, with additional limitations (see non-functional requirements).
- Object files compatible with the latest version of **cocas**.
- Assembly files compatible with the latest version of **cocas**.

Object files are not processed when generating an assembly code file (see “Output files” section).

5.1.2. Output files

The compiler shall be able to produce files of the following types:

- CdM-16 assembly files compatible with the latest version of **cocas**;
- Object files compatible with the latest version of **cocas**.
- LLVM IR text files;
- LLVM bitcode files.

5.1.3. Supported options

The following options are supported as specified in “Clang command line argument reference”:

- general flags (listed in “introduction”);
- action flags;
- preprocessor options;
- optimization level flags.

See [Clang command line argument reference](#).

5.2. Use cases for Rustc users

Use case "compile with rustc"

The usage of the CdM LLVM backend for compiling rust code with the Rustc compiler.

Actor: Rustc user

Goals: To process one or many input files and produce one or many output files (see “Input files” and “Output files” sections).

Precondition: The user has one or more input files.

Main success scenario

Steps:

1. User executes the program providing the necessary options and the names of the input files in command line arguments.
2. The program prints any warning messages to `stderr`.
3. The program writes one or more output files.
4. The program exits with code 0.

Alternative scenario "compilation error"

Conditions - any of:

- The provided combination of options is not supported (see “Supported options” section).
- Some of the input files do not exist at the specified locations.
- Some of the input files are invalid (see “Input files” section).
- No input files were specified.

Steps:

1. User executes the program providing the necessary options and the names of the input files in command line arguments.
2. The program prints the error and warning messages to `stderr`.
3. The program writes zero or more output files not affected by the errors.
4. The program exits with some code other than 0.

5.2.1. Input files

The compiler shall accept any combination of files of the following types:

- Rust source files, as specified in the Rust language reference, with additional limitations (see non-function requirements).

5.2.2. Output files

The compiler shall be able to produce files of the following types:

- CdM-16 assembly files
- Linkable object files and libraries
- LLVM IR text files.
- LLVM bitcode files.

5.3. Common use cases

Use case "print compiler information"

Getting information about the compilation environment.

Actor: Clang or rustc user

Goals: To get some information about LLVM or the compiler (clang or rustc).

Main success scenario

Steps:

1. User executes the program with some informational flags (see the CLI reference for the corresponding compiler).
2. The program ignores all compilation-related options and prints the requested information to `stdout`.
3. The program exits with code 0.

Alternative scenario "syntax error"

Condition: the provided combination of options is not supported

Steps:

1. User executes the program.
2. The program prints the error and warning messages to `stderr`.
3. The program exits with some code other than 0.

6. Non-functional requirements

6.1. Environment

- OS: Ubuntu 24.04 / MacOS 26
- Software: cocas (latest)

6.2. Language support limitations

6.2.1. ISO/GNU C Language

As CdM-16 is by default an unhosted platform, we provide only freestanding conformance to the C language standard as defined by clauses 4.6 and 5.1.2 of ISO/IEC 9899:2024.

Using unsupported features must result in error at some point of compilation, assembly, or linkage pipeline.

We do not provide support for floating point operations and atomics.

6.2.2. Rust programming language

For Rust, we only provide support for features that are sensible for a `no_std` single-processor freestanding execution environment.

This includes Rust's [core](#) and [alloc](#) crates, with the exception of atomic operations.

6.3. Extensibility

Code for CdM-16 target support shall be as decoupled as possible from mainline LLVM. This should be achieved by implementing target-specific classes and modules, using adjustments to common code only when necessary.